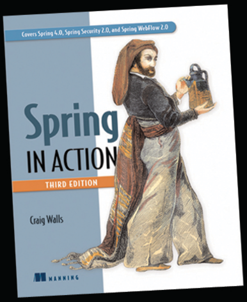
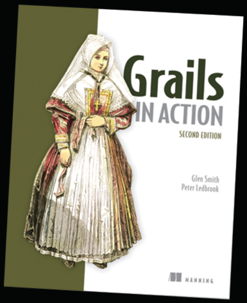

Covers Spring 4.0, Spring Security 2.0, and Spring WebFlow 2.0
Spring
IN ACTION
THIRD EDITION
Craig Walls
MANNING



Spring does *that*?

A sampler on Spring and related technologies from Manning Publications authors

MANNING

Spring does *that*?

A sampler on Spring and related technologies from Manning Publications authors

Edited by Ken Rimple

MANNING
Shelter Island

Spring does *that*?

A special edition eBook

contents

about the Spring framework and its ecosystem

Agile and adaptive development on the leading edge

by Ken Rimple, author of *Spring Roo in Action*

It's been more than ten years since Rod Johnson first conceived of the highly productive Spring Framework. At the time, Java programmers were taught to "program"and then "configure" their applications using special build scripts, EJB compilers, and somewhat (but not completely) compatible application servers.

The Spring framework eliminated the ceremony required when developing applications on the Enterprise Java platform: no more compiling and packaging tedious EJBs or EAR files, dealing with mostly tedious deployment descriptors, or getting hung up on platform-specific issues or vendor lock-in. In fact, Spring code can be tested within a JUnit system test, executed within a Java main method, in a servlet container, or even behind your EJBs.

The reason Spring has such staying power more than ten years later is because it consistently and efficiently simplifies programming tasks by eliminating boilerplate. Spring developers hand-configure as little as they can, try to eschew state where possible, and let the wiring of components take place outside of the code itself. Your components are essentially simple Java interfaces and classes, and any external API features are given to you through dependency injection.

Spring programmers know that if they are looking to integrate with a particular technology, they should check to see if there is a Spring Enterprise configuration component available for it. They also look for any Spring-provided helper APIs, such as templates, which may make it easier to issue complex but repetitive sets of calls, such as sending and receiving JMS messages, calling a RESTful web service, or sending an email.

Finally, for more complex configurations, Spring provides domain-specific configuration helpers in the form of JavaConfig components, or using XML, Scala, or Groovy-based DSLs. The Spring Security framework, Spring Integration, and even Grails are places where you'll see this technique applied.

Over the years, a number of projects have emerged built on top of the Spring framework: Spring MVC, Spring Web Flow, Spring Integration, Spring Batch, and Spring Security, to name a few. All of these projects run as Spring beans and infrastructure APIs. None of them are proprietary. They all live in GitHub, available for pull requests whenever a contributer finds a bug and wants to donate a fix. Many of these projects have Manning In Action books dedicated to them. A number of other technologies, such as ActiveMQ, are written using Spring itself.

It is heartening to see Javascript frameworks like AngularJS embrace dependency injection natively. Java EE itself has adopted the Container Dependency Injection specification, contributed to by Rod Johnson, the innovator who ended up bringing simplification back into the container from which he originally wanted to run screaming.

This sampler will explore some of the more interesting features of the Spring Framework and Spring-based APIs. We hope to show the scope of what is available to you to help integrate and simplify your applications and take advantage of some of the more modern APIs and tools coming out of the developers who developed the Spring container API all of those years ago.

about this eBook

This sampler consists of six excerpts taken from five of Manning's bestselling Spring books, selected by Ken Rimple. Click on the titles below to learn more or to purchase.

Spring in Action, Third Edition
by Craig Walls
Published June 2011

Spring Roo in Action
by Ken Rimple and Srini Penchikala
Published April 2012

Spring Integration in Action
by Mark Fisher, Jonas Partner, Marius Bogoevici, and Iwein Fuld
Published September 2012

Spring Batch in Action
by Arnaud Cogoluegnes, Thierry Templier, Gary Gregory, Olivier Bazoud
Published September 2010

Grails in Action, Second Edition
by Glen Smith and Peter Ledbrook
Published September 2012

about the authors

KEN RIMPLE is a veteran Java developer, trainer, mentor, and head of Chariot's Education Services team, a VMWare training partner. He lives in the Philadelphia area. SRINI PENCHIKALA is a security architect with over 16 years of experience in software design and development. He lives in Austin, Texas.

CRAIG WALLS is a software developer at SpringSource. He's a popular author who has written four previous books for Manning, and a frequent speaker at user groups and conferences. Craig lives in Plano, Texas.

A Java EE architect, ARNAUD COGOLUÈGNES specializes in middleware. THIERRY TEMPLIER is a Java EE and rich web architect. He contributed the JCA and Lucene to Spring. ANDY PIPER is a software architect with Oracle and a committer on the Spring DM project.

ARNAUD COGOLUEGNES, THIERRY TEMPLIER, and OLIVIER BAZOUD are Java EE architects with a focus on Spring. GARY GREGORY is a Java developer and software integration specialist.

A frequent speaker and the co-host of the Grails podcast, GLEN SMITH launched the first public-facing Grails app (an SMS Gateway) on Grails 0.2. PETER LEDBROOK is a core Grails developer and author of several popular plugins, who has worked as an engineer for both G2One and SpringSource.

Spring and JPA

From *Spring in Action, Third Edition*
by Craig Walls

Spring's biggest strengths lie in its integration with various databases. A good place to start researching database access APIs is JPA. This excerpt from Spring in Action, Third Edition, details how to configure and use Spring with the JPA API.

From its beginning, the EJB specification has included the concept of entity beans. In EJB, *entity beans* are a type of EJB that describes business objects that are persisted in a relational database. Entity beans have undergone several tweaks over the years, including *bean-managed persistence (BMP)* entity beans and *container-managed persistence (CMP)* entity beans.

Entity beans both enjoyed the rise and suffered the fall of EJB's popularity. In recent years, developers have traded in their heavyweight EJBs for simpler POJO-based development. This presented a challenge to the Java Community Process to shape the new EJB specification around POJOs. The result is JSR-220—also known as *EJB 3*.

The Java Persistence API (JPA) emerged out of the rubble of EJB 2's entity beans as the next-generation Java persistence standard. JPA is a POJO-based persistence mechanism that draws ideas from both Hibernate and *Java Data Objects (JDO)*, and mixes Java 5 annotations in for good measure.

With the Spring 2.0 release came the premiere of Spring integration with JPA. The irony is that many blame (or credit) Spring with the demise of EJB. But now that Spring provides support for JPA, many developers are recommending JPA for persistence in Spring-based applications. In fact, some say that Spring-JPA is the dream team for POJO development.

The first step toward using JPA with Spring is to configure an entity manager factory as a bean in the Spring application context.

Configuring an entity manager factory

In a nutshell, JPA-based applications use an implementation of `EntityManager-Factory` to get an instance of an `EntityManager`. The JPA specification defines two kinds of entity managers:

- *Application-managed*—Entity managers are created when an application directly requests one from an entity manager factory. With application-managed entity managers, the application is responsible for opening or closing entity managers and involving the entity manager in transactions. This type of entity manager is most appropriate for use in standalone applications that don't run within a Java EE container.
- *Container-managed*—Entity managers are created and managed by a Java EE container. The application doesn't interact with the entity manager factory at all. Instead, entity managers are obtained directly through injection or from JNDI. The container is responsible for configuring the entity manager factories. This type of entity manager is most appropriate for use by a Java EE container that wants to maintain some control over JPA configuration beyond what's specified in persistence.xml.

Both kinds of entity manager implement the same `EntityManager` interface. The key difference isn't in the `EntityManager` itself, but rather in how the `EntityManager` is created and managed. Application-managed `EntityManagers` are created by an `Entity-ManagerFactory` obtained by calling the `createEntityManagerFactory()` method of the `PersistenceProvider`. Meanwhile, container-managed `EntityManagerFactorys` are obtained through `PersistenceProvider`'s `createContainerEntityManager-Factory()` method.

So what does this all mean for Spring developers wanting to use JPA? Not much. Regardless of which variety of `EntityManagerFactory` you want to use, Spring will take responsibility for managing `EntityManagers` for you. If you're using an application-managed entity manager, Spring plays the role of an application and transparently deals with the `EntityManager` on your behalf. In the container-managed scenario, Spring plays the role of the container.

Each flavor of entity manager factory is produced by a corresponding Spring factory bean:

- `LocalEntityManagerFactoryBean` produces an application-managed `Entity-ManagerFactory`.
- `LocalContainerEntityManagerFactoryBean` produces a container-managed `EntityManagerFactory`.

It's important to point out that the choice made between an application-managed `EntityManagerFactory` and a container-managed `EntityManagerFactory` is completely transparent to a Spring-based application. Spring's `JpaTemplate` hides the intricate details of dealing with either form of `EntityManagerFactory`, leaving your data access code to focus on its true purpose: data access.

The only real difference between application-managed and container-managed entity manager factories, as far as Spring is concerned, is how each is configured within the Spring application context. Let's start by looking at how to configure the application-managed `LocalEntityManagerFactoryBean` in Spring. Then we'll see how to configure a container-managed `LocalContainerEntityManagerFactoryBean`.

Configuring application-managed JPA

Application-managed entity manager factories derive most of their configuration information from a configuration file called persistence.xml. This file must appear in the META-INF directory within the classpath.

The purpose of the persistence.xml file is to define one or more persistence units. A persistence unit is a grouping of one or more persistent classes that correspond to a single data source. In simple terms, persistence.xml enumerates one or more persistent classes along with any additional configuration such as data sources and XML-based mapping files. Here's a typical example of a persistence.xml file as it pertains to the Spitter application:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    version="1.0">
  <persistence-unit name="spitterPU">
    <class>com.habuma.spitter.domain.Spitter</class>
    <class>com.habuma.spitter.domain.Spittle</class>
    <properties>
      <property name="toplink.jdbc.driver"
          value="org.hsqldb.jdbcDriver" />
      <property name="toplink.jdbc.url" value=
          "jdbc:hsqldb:hsql://localhost/spitter/spitter" />
      <property name="toplink.jdbc.user"
          value="sa" />
      <property name="toplink.jdbc.password"
          value="" />
    </properties>
  </persistence-unit>
</persistence>
```

Because so much configuration goes into a persistence.xml file, little configuration is required (or even possible) in Spring. The following `<bean>` declares a `LocalEntity-ManagerFactoryBean` in Spring:

```
<bean id="emf"
      class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="spitterPU" />
</bean>
```

The value given to the `persistenceUnitName` property refers to the persistence unit name as it appears in persistence.xml.

The reason why much of what goes into creating an application-managed `Entity-ManagerFactory` is contained in persistence.xml has everything to do with what it means to be application-managed. In the application-managed scenario (not involving Spring), an application is entirely responsible for obtaining an `EntityManagerFactory` through the JPA implementation's PersistenceProvider. The application code would become incredibly bloated if it had to define the persistence unit every time it requested an EntityManagerFactory. By specifying it in persistence.xml, JPA can look in this well-known location for persistence unit definitions.

But with Spring's support for JPA, we'll never deal directly with the Persistence-Provider. Therefore, it seems silly to extract configuration information into persistence.xml. In fact, doing so prevents us from configuring the EntityManagerFactory in Spring (so that, for example, we can provide a Spring-configured data source).

For that reason, we should turn our attention to container-managed JPA.

Configuring container-managed JPA

Container-managed JPA takes a different approach. When running within a container, an EntityManagerFactory can be produced using information provided by the container—Spring, in our case.

Instead of configuring data source details in persistence.xml, you can configure this information in the Spring application context. For example, the following <bean> declaration shows how to configure container-managed JPA in Spring using LocalContainerEntityManagerFactoryBean.

```
<bean id="emf" class=
      "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
</bean>
```

Here we've configured the dataSource property with a Spring-configured data source. Any implementation of javax.sql.DataSource is appropriate. Although a data source may still be configured in persistence.xml, the data source specified through this property takes precedence.

The jpaVendorAdapter property can be used to provide specifics about the particular JPA implementation to use. Spring comes with a handful of JPA vendor adaptors to choose from:

- EclipseLinkJpaVendorAdapter
- HibernateJpaVendorAdapter
- OpenJpaVendorAdapter
- TopLinkJpaVendorAdapter

In this case, we're using Hibernate as a JPA implementation, so we've configured it with a HibernateJpaVendorAdapter:

```
<bean id="jpaVendorAdapter"
      class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
  <property name="database" value="HSQL" />
  <property name="showSql" value="true"/>
  <property name="generateDdl" value="false"/>
  <property name="databasePlatform"
            value="org.hibernate.dialect.HSQLDialect" />
</bean>
```

Database platform	Value for `database` property
IBM DB2	`DB2`
Apache Derby	`DERBY`
H2	`H2`
Hypersonic	`HSQL`
Informix	`INFORMIX`
MySQL	`MYSQL`
Oracle	`ORACLE`
PostgreSQL	`POSTGRESQL`
Microsoft SQL Server	`SQLSERVER`
Sybase	`SYBASE`

Table 1 The Hibernate JPA vendor adapter supports several databases. You can specify which database to use by setting its `database` property.

Several properties are set on the vendor adapter, but the most important one is the `database` property, where we've specified the Hypersonic database as the database we'll be using. Other values supported for this property include those listed in table 1.

Certain dynamic persistence features require that the class of persistent objects be modified with instrumentation to support the feature. Objects whose properties are lazily loaded (they won't be retrieved from the database until they're accessed) must have their class instrumented with code that knows to retrieve unloaded data upon access. Some frameworks use dynamic proxies to implement lazy loading. Others, such as JDO, perform class instrumentation at compile time.

Which entity manager factory bean you choose will depend primarily on how you'll use it. For simple applications, `LocalEntityManagerFactoryBean` may be sufficient. But because `LocalContainerEntityManagerFactoryBean` enables us to configure more of JPA in Spring, it's an attractive choice and likely the one that you'll choose for production use.

Pulling an EntityManagerFactory from JNDI

It's also worth noting that if you're deploying your Spring application in some application servers, an `EntityManagerFactory` may have already been created for you and may be waiting in JNDI to be retrieved. In that case, you can use the `<jee:jndi-lookup>` element from Spring's `jee` namespace to nab a reference to the `EntityManagerFactory`:

```
<jee:jndi-lookup id="emf" jndi-name="persistence/spitterPU" />
```

Regardless of how you get your hands on an `EntityManagerFactory`, once you have one, you're ready to start writing a DAO. Let's do that now.

Writing a JPA-based DAO

Just like all of Spring's other persistence integration options, Spring-JPA integration comes in template form with `JpaTemplate` and a corresponding `JpaDaoSupport` class. Nevertheless, template-based JPA has been set aside in favor of a pure JPA approach.

Since pure JPA is favored over template-based JPA, we'll focus on building Spring-free JPA DAOs in this section. Specifically, `JpaSpitterDao` in the following listing shows how to develop a JPA DAO without resorting to using Spring's `JpaTemplate`.

Listing 1 A pure JPA DAO doesn't use any Spring templates.

```
package com.habuma.spitter.persistence;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.dao.DataAccessException;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;

@Repository("spitterDao")
@Transactional
public class JpaSpitterDao implements SpitterDao {
  private static final String RECENT_SPITTLES =
      "SELECT s FROM Spittle s";
  private static final String ALL_SPITTERS =
      "SELECT s FROM Spitter s";
  private static final String SPITTER_FOR_USERNAME =
      "SELECT s FROM Spitter s WHERE s.username = :username";
  private static final String SPITTLES_BY_USERNAME =
      "SELECT s FROM Spittle s WHERE s.spitter.username = :username";

  @PersistenceContext                            Inject
  private EntityManager em;                      EntityManager

  public void addSpitter(Spitter spitter) {
    em.persist(spitter);
  }

  public Spitter getSpitterById(long id) {                  Use
    return em.find(Spitter.class, id);                      EntityManager
  }

  public void saveSpitter(Spitter spitter) {
    em.merge(spitter);
  }
...
}
```

`JpaSpitterDao` uses an `EntityManager` to handle persistence. By working with an `EntityManager`, the DAO remains pure and resembles how a similar DAO may appear in a non-Spring application. But where does it get the `EntityManager`?

Note that the em property is annotated with @PersistentContext. Put plainly, that annotation indicates that an instance of EntityManager should be injected into em. To enable EntityManager injection in Spring, we'll need to configure a Persistence-AnnotationBeanPostProcessor in Spring's application context:

```
<bean class="org.springframework.orm.jpa.support.
   ➥PersistenceAnnotationBeanPostProcessor"/>
```

You may have also noticed that JpaSpitterDao is annotated with @Repository and @Transactional. @Transactional indicates that the persistence methods in this DAO will be involved in a transactional context.

As for @Repository, it serves the same purpose here as it did when we developed the Hibernate contextual session version of the DAO. Without a template to handle exception translation, we need to annotate our DAO with @Repository so that PersistenceExceptionTranslationPostProcessor will know that this is one of those beans for which exceptions should be translated into one of Spring's unified data access exceptions.

Speaking of PersistenceExceptionTranslationPostProcessor, we'll need to remember to wire it up as a bean in Spring just as we did for the Hibernate example:

```
<bean class="org.springframework.dao.annotation.
  ➥PersistenceExceptionTranslationPostProcessor"/>
```

Note that exception translation, whether it be with JPA or Hibernate, isn't mandatory. If you'd prefer that your DAO throw JPA-specific or Hibernate-specific exceptions, then you're welcome to forgo PersistenceExceptionTranslationPostProcessor and let the native exceptions flow freely. But if you do use Spring's exception translation, you'll be unifying all of your data access exceptions under Spring's exception hierarchy, which will make it easier to swap out persistence mechanisms later.

Summary

We saw how to build the persistence layer of a Spring application using Java Persistence API (JPA) . Some other options are Java Database Connectivity (JDBC) and Hibernate. Which you choose is largely a matter of taste, but because we developed our persistence layer behind a common Java interface, the rest of our application can remain unaware of how data is ferried to and from the database.

JPA setup and DAO

From *Spring Roo in Action*
by Ken Rimple and Srini Penchikala

Spring Roo has a good deal of support for testing JPA out of the box and can be used as a great tool to design your database schemas. Chapters 3 and 4 of the book cover entities and relationships. This excerpt from chapter 3 explains how the Spring Data API can be used to simplify JPA calls.

What if you don't like the approach of encapsulating your JPA code within each entity? Perhaps you have a more complex model, one where the boundaries for queries and transactions are a bit more blurred, and some of the code fits best when manipulating or querying more than one entity at a time? If this is your situation, or if you prefer a layered approach that separates the data logic from your entity classes, you can tell Roo to build JPA repositories for you.

Roo repositories are built using the relatively new Spring Data API. Spring Data provides support for dynamically generated proxy classes for a given entity, and those classes handle all of the methods you're used to coding by hand (or using in the Active Record entities).

It is quite easy to generate a repository. Let's build a repository to back the `Course` entity:

```
repository jpa --interface ~.db.CourseRepository ➥
     --entity ~.model.Course
```

This command generates a repository class:

```
package org.rooinaction.coursemanager.db;

import org.rooinaction.rooinaction.coursemanager.model.Course;
import org.springframework.roo.addon.layers.repository➥
          .jpa.RooJpaRepository;

@RooJpaRepository(domainType = Course.class)
```

```
public interface CourseRepository {
}
```

There are no methods defined in this interface; it exists merely as a holding place for the `@RooJpaRepository` annotation. The interface *is* backed by an ITD. In this case, the file is named `CourseRepository_Roo_Repository.aj`:

```
package org.rooinaction.rooinaction.coursemanager.db;

import java.lang.Long;
import org.rooinaction.rooinaction.coursemanager.model.Course;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;
import org.springframework.stereotype.Repository;

privileged aspect CourseRepository_Roo_Jpa_Repository {

  declare parents: CourseRepository ➥
    extends JpaRepository<Course, Long>;

  declare parents: CourseRepository ➥
    extends JpaSpecificationExecutor<Course>;

  declare @type: CourseRepository: @Repository;

}
```

These two files may be a bit baffling to you if you're used to coding your own repositories. Roo uses the typical Spring pattern of annotating the repository with `@Repository`, which marks it as a Spring bean and provides exception translation, but it also extends it with two additional interfaces—`JpaRepository` and `JpaSpecificationExecutor`. Let's take a look at each one, starting with `JpaRepository`.

The JpaRepository API

Look at the methods implemented by the `JpaRepository` class:

```
java.util.List<T> findAll();
java.util.List<T> findAll(org.springframework.data.domain.Sort sort);
java.util.List<T> save(java.lang.Iterable<? extends T> iterable);
void flush();
T saveAndFlush(T t);
void deleteInBatch(java.lang.Iterable<T> tIterable);
```

These are all methods to search, save, and remove data from the entity. Note that the `<T>` designation is a Java generic type. Since the `CourseRepository` is defined as implementing `JpaRepository<Course, Long>`, all of the generic `<T>` methods will take `Course` entities as arguments, and expect a `Long`-based primary key.

Let's test this API using a JUnit test. Add the following test to your `CourseIntegrationTest` class:

```
@Test
@Transactional
public void addAndFetchCourseViaRepo() {
  Course c = new Course();
  c.setCourseType(CourseTypeEnum.CONTINUING_EDUCATION);
  c.setName("Stand-up Comedy");
  c.setDescription(
    "You'll laugh, you'll cry, it will become a part of you.");
  c.setMaximumCapacity(10);

  courseRepository.saveAndFlush(c);
  c.clear();

  Assert.assertNotNull(c.getId());

  Course c2 = courseRepository.findOne(c.getId());
  Assert.assertNotNull(c2);
  Assert.assertEquals(c.getName(), c2.getName());
  Assert.assertEquals(c2.getDescription(), c.getDescription());
  Assert.assertEquals(
    c.getMaximumCapacity(), c2.getMaximumCapacity());
  Assert.assertEquals(c.getCourseType(), c2.getCourseType());
}
```

So now you can use a Roo repository to implement your JPA code. The methods `saveAndFlush()` and `getOne(Long)` are provided dynamically at runtime via the Spring Data API.

Queries with JpaSpecificationImplementor

But wait, there are more features to explore here. What does the second interface, `JpaSpecificationImplementor`, provide?

```
T findOne(Specification<T> tSpecification);
List<T> findAll(Specification<T> tSpecification
Page<T> findAll(Specification<T> tSpecification, Pageable pageable);
List<T> findAll(Specification<T> tSpecification, Sort sort);
long count(Specification<T> tSpecification);
```

This interface provides access to the Spring Data features for providing criteria-based query and paging support. The methods accept a `Specification` class, which is used to define the search criteria to pass to the repository to find, sort, and page through a list of entities, or fetch a single entity. For example, to provide a predicate that expects a non-null run date:

```
public class CourseSpecifications {

  public static Specification<Course> hasRunDate() {

    return new Specification<Course>() {
      @Override
      public Predicate toPredicate(
          Root<Course> root,
           CriteriaQuery<?> query,
          CriteriaBuilder cb) {                    Exposes field types
        return cb.isNotNull(
          root.get("runDate"));                    Literate API
      }
    };
  }
}
```

The `toPredicate()` method takes a `Root<Course>`, which provides access to the types in the JPA entity, a JPA `CriteriaQuery`, which is built by Spring and passed into the method automatically at runtime to be executed, and a `CriteriaBuilder`, which allows you to add predicates to the query using English language–like calls, such as `cb.isNotNull` above.

To use the specification, you just need to call the static `CourseSpecifications.hasRunDate()` method, and pass it to the appropriate finder:

```
List<Course> courses = courseRepository.findAll(
                           CourseSpecifications.hasRunDate());
```

This approach is similar to writing criteria-based JPA queries, but is in marked contrast to Roo finders, which are attached normally to Active Record entities annotated with `@RooJpaActiveRecord`.

Annotation-driven queries with @Query

One of the most powerful features of the Spring Data JPA API is providing annotation-driven queries. Since Spring Data builds the implementation class at runtime, you can define methods in your interface that Roo can use to implement custom queries and even updates.

Let's look at an example method. You can define a query method in your `CourseRepository` interface to find all student registrations for a given student and date range:

```
@Query("select distinct r from Registration as r " +
    "where r.student.id = :studentId " +
    "and r.offering.offerDate between :start and :end")

@Transactional(readOnly = true)
List<Registration> findStudentRegistrationForOfferingsInDateRange(
      @Param("studentId") long studentId,
      @Param("start") Date start,
      @Param("end") Date end);
```

Roo implements the code for this method at runtime, based on the Spring Data `@Query` annotation. All parameters in the example above are defined using the `@Param` annotation, and the type returned is defined as the return type of the method, `List<Registration>`. Note that you've also passed the `@Transactional` annotation, and marked the query as a read-only transaction.

You can perform updates using the `@Query` method as well, as long as you mark the method as `@Modifying`:

```
@Query("update Registration r set attended = :attended " +
       "where r.student.id = :studentId")
@Modifying
@Transactional
void updateAttendance(
      @Param("studentId") long studentId,
      @Param("attended") boolean attended);
```

In this example, you've marked your interface method with `@Modifying` to signify that you're expecting a data manipulation statement, not just a simple `SELECT` statement. You also define your method with `@Transactional`, so that it's wrapped with a read/write transaction.

Spring Roo builds the implementation classes automatically, based on a Spring configuration file in META-INF/spring named applicationContext-jpa.xml. This file contains the Spring Data XML configuration element, `<repositories/>`, which scans for and mounts interface-driven repositories:

```
<repositories base-package="org.rooinaction.coursemanager" />
```

The package defined in this Spring XML configuration element is your root project package. You can now add repositories in whatever subpackage makes sense. You don't have to use Roo to generate your Spring Data classes either, so if you're already a Spring Data or JPA expert, just code away!

For more about the Spring Data JPA API, visit the project website at http://mng.bz/63xp.

Summary

As you've seen, you can use repositories in a more traditional Spring layered application instead of applying the Active Record pattern. Roo even rewrites your automated entity integration tests automatically, when it detects that you've added a repository for a given entity. You can always fall back to the typical interface-and-implementation JPA repository where necessary.

As an added bonus, you can skip the Active Record generation for Roo entities by issuing the `--activeRecord false` attribute when defining an entity:

```
roo> entity jpa --class ~.model.Course --activeRecord false
```

IF YOU'VE BEEN USING ACTIVE RECORD AND WANT TO MIGRATE... Just edit your entity, and replace `@RooJpaActiveRecord` with `@RooJpaEntity`. Fire up the Roo shell and watch it remove all of those Active Record ITDs. Follow up by creating a JPA repository and you're all set. If you take advantage of Roo's web interface scaffolding, Roo will even reroute calls in the controller to the repository after you create one.

Spring and integration

from *Spring Integration in Action*
by Mark Fisher, Jonas Partner,
Marius Bogoevici, and Iwein Fuld

A well-established system integration approach involves the use of middleware that implements Gregor and Wolfe's Enterprise Integration Patterns, documented in the book by the same name. Spring Integration is a message-oriented middleware platform built atop the Spring container and using EIP terms for configuring components such as routers, hubs, wire-taps, load balancers, and other features.

This excerpt shows how to configure communication channels, a foundational component conveying data to parts of your application.

Messages don't achieve anything by sitting there all by themselves. To do something useful with the information they're packaging, they need to travel from one component to another, and for this they need *channels*, which are well-defined conduits for transporting messages across the system.

Let's use a letter analogy. The sender creates the letter and hands it off to the mailing system by depositing it in a well-known location: the mailbox. From there on, the letter is completely under the control of the mailing system, which delivers it to various waypoints until it reaches the recipient. The most that the sender can expect is a reply. The sender is unaware of who routes the message or, sometimes, even who may be the physical reader of the letter (think about writing to a government agency). From a logical standpoint, the channel is much like a mailbox: a place where components (producers) deposit messages that are later processed by other components (consumers). This way, producers and consumers are decoupled from each other and are only concerned about what kinds of messages they can send and receive, respectively.

One distinctive trait of Spring Integration, which differentiates it from other enterprise integration frameworks, is its emphasis on the role of channels in defining the enterprise integration strategy. Channels aren't just information transfer components; they play an active role in defining the overall application behavior. The business processing takes place in the endpoints, but you can alter the channel configuration to completely change the runtime characteristics of the application.

We explain channels from a logical perspective and offer overviews of the various channel implementations provided by the framework: what's characteristic to each of them, and how you can get the most from your application by using the right kind of channel for the job.

Using channels to move messages

To connect the producers and consumers configured in an application, you use a channel. All channels in Spring Integration implement the following `MessageChannel` interface, which defines standard methods for sending messages. Note that it provides no methods for receiving messages:

```
package org.springframework.integration;
public interface MessageChannel {
    boolean send(Message<?> message);
    boolean send(Message<?> message, long timeout);
}
```

The reason no methods are provided for receiving messages is because Spring Integration differentiates clearly between two mechanisms through which messages are handed over to the next endpoint—polling and subscription—and provides two distinct types of channels accordingly.

I'll let you know when I've got something!

Channels that implement the `SubscribableChannel` interface, shown below, take responsibility for notifying subscribers when a message is available:

```
package org.springframework.integration.core;
public interface SubscribableChannel extends MessageChannel {
    boolean subscribe(MessageHandler handler);
    boolean unsubscribe(MessageHandler handler);
}
```

Do you have any messages for me?

The alternative is the `PollableChannel`, whose definition follows. This type of channel requires the receiver or the framework acting on behalf of the receiver to periodically check whether messages are available on the channel. This approach has the advantage that the consumer can choose when to process messages. The approach can also have its downsides, requiring a trade-off between longer poll periods, which may introduce latency in receiving a message, and computation overhead from more frequent polls that find no messages:

```
package org.springframework.integration.core;

public interface PollableChannel extends MessageChannel {

    Message<?> receive();

    Message<?> receive(long timeout);
}
```

It's important to understand the characteristics of each message delivery strategy because the decision to use one over the other affects the timeliness and scalability of the system. From a logical point of view, the responsibility of connecting a consumer to a channel belongs to the framework, thus alleviating the complications of defining the appropriate consumer types. To put it plainly, your job is to configure the appropriate channel type, and the framework will select the appropriate consumer type (polling or event-driven).

Also, subscription versus polling is the most important criterion for classifying channels, but it's not the only one. In choosing the right channels for your application, you must consider a number of other criteria, which we discuss next.

The right channel for the job

Spring Integration offers a number of channel implementations, and because `MessageChannel` is an interface, you're also free to provide your own implementations. The type of channel you select has significant implications for your application, including transactional boundaries, latency, and overall throughput. This section walks you through the factors to consider and through a practical scenario for selecting appropriate channels. In the configuration, we use the namespace, and we also discuss which concrete channel implementation will be instantiated by the framework.

In our flight-booking internet application, a booking confirmation results in a number of actions. Foremost for many businesses is the need to get paid, so making sure you can charge the provided credit card is a high priority. You also want to ensure that, as seats are booked, an update occurs to indicate one less seat is available on the flight so you don't overbook the flight. The system must also send a confirmation email with details of the booking and additional information on the check-in process. In addition to a website, the internet booking application exposes a REST interface to allow third-party integration for flight comparison sites and resellers. Because most of the airline's premium customers come through the airline's website, any design should allow you to prioritize bookings originating from its website over third-party integration requests to ensure that the airline's direct customers experience a responsive website even during high load.

The selection of channels is based on both functional and nonfunctional requirements, and several factors can help you make the right choice. Table 1 provides a brief overview of the technical criteria and the best practices you should consider when selecting the most appropriate channels.

Let's see how these criteria apply to our flight-booking sample.

Table 1 How do you decide what channel to use?

Decision factor	What factors must you consider?
Sharing context	– Do you need to propagate context information between the successive steps of a process? – Thread-local variables are used to propagate context when needed in several places where passing via the stack would needlessly increase coupling, such as in the transaction context. – Relying on the thread context is a subtle form of coupling and has an impact when considering the adoption of a highly asynchronous staged event-driven architecture (SEDA) model. It may prevent splitting the processing into concurrently executing steps, prevent partial failures, or introduce security risks such as leaking permissions to the processing of different messages.
Atomic boundaries	– Do you have all-or-nothing scenarios? – Classic example: bank transaction where credit and debit should either both succeed or both fail. – Typically used to decide transaction boundaries, which makes it a specific case of context sharing. Influences the threading model and therefore limits the available options when choosing a channel type.
Buffering messages	– Do you need to consider variable load? What is immediate and what can wait? – The ability of systems to withstand high loads is an important performance factor, but load is typically fluctuating, so adopting a thread-per-message-processing scenario requires more hardware resources for accommodating peak load situations. Those resources are unused when the load decreases, so this approach could be expensive and inefficient. Moreover, some of the steps may be slow, so resources may be blocked for long durations. – Consider what requires an immediate response and what can be delayed; then use a buffer to store incoming requests at peak rate, and allow the system to process them at its own pace. Consider mixing the types of processing—for example, an online purchase system that immediately acknowledges receipt of the request, performs some mandatory steps (credit card processing, order number generation), and responds to the client but does the actual handling of the request (assembling the items, shipping, and so on) asynchronously in the background.
Blocking and nonblocking operations	– How many messages can you buffer? What should you do when you can't cope with demand? – If your application can't cope with the number of messages being received and no limits are in place, you may exhaust your capacity for storing the message backlog or breach quality-of-service guarantees in terms of response turnaround. – Recognizing that the system can't cope with demands is usually a better option than continuing to build up a backlog. A common approach is to apply a degree of self-limiting behavior to the system by blocking the acceptance of new messages when the system is approaching its maximum capacity. This limit commonly is a maximum number of messages awaiting processing or a measure of requests received per second. – Where the requester has a finite number of threads for issuing requests, blocking those threads for long periods of time may result in timeouts or quality-of-service breaches. It may be preferable to accept the message and then discard it later if system capacity is being exceeded or to set a timeout on the blocking operation to avoid indefinite blocking of the requester.

Table 1 How do you decide what channel to use? *(continued)*

Decision factor	What factors must you consider?
Consumption model	– How many components are interested in receiving a particular message? – There are two major messaging paradigms: point-to-point and publish-subscribe. In the former, a message is consumed by exactly one recipient connected to the channel (even if there are more of them), and in the latter, the message is received by all recipients. – If the processing requirements are that the same message should be handled by multiple consumers, the consumers can work concurrently and a publish-subscribe channel can take care of that. An example is a mashup application that aggregates results from searching flight bookings. Requests are broadcast simultaneously to all potential providers, which will respond by indicating whether they can offer a booking. – Conversely, if the request should always be handled by a single component (for example, for processing a payment), you need a point-to-point strategy.

A channel selection example

Using the default channel throughout, we have three channels—one accepting requests and the other two connecting the services:

```
<channel id="bookingConfirmationRequests"/>

<service-activator input-channel="bookingConfirmationRequests"
                   output-channel="chargedBookings"
                   ref="billForBookingService" />

<channel id="chargedBookings" />

<service-activator input-channel="chargedBookings"
                   output-channel="emailConfirmationRequests"
                   ref="seatAvailabilityService" />

<channel id="emailConfirmationRequests" />

<outbound-channel-adapter channel="emailConfirmationRequests"
                          ref="emailConfirmationService" />
```

In Spring Integration, the default channels are `SubscribableChannels`, and the message transmission is synchronous. The effect is simple: one thread is responsible for invoking the three services sequentially, as shown in figure 1.

Because all operations are executing in a single thread, a single transaction encompasses those invocations. That assumes that the transaction configuration doesn't require new transactions to be created for any of the services.

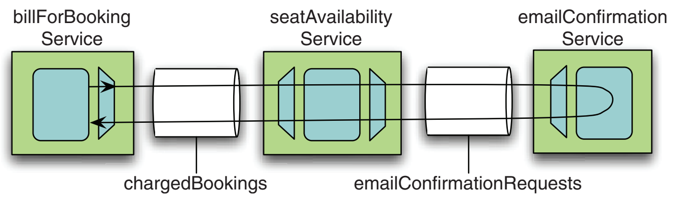


Figure 1 Diagram of threading model of service invocation in the airline application

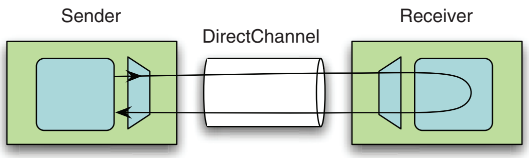


Figure 2 Diagram of threading model of service invocation when using a default channel

Figure 2 shows what you get when you configure an application using the default channels, which are subscribable and synchronous. But having all service invocations happening in one thread and encompassed by a single transaction is a mixed blessing: it could be a good thing in applications where all three operations must be executed atomically, but it takes a toll on the scalability and robustness of the application.

But email is slow and our servers are unreliable

The basic configuration is good in the sunny-day case where the email server is always up and responsive, and the network is 100% reliable. Reality is different. Your application needs to work in a world where the email server is sometimes overloaded and the network sometimes fails. Analyzing your actions in terms of what you need to do now and what you can afford to do later is a good way of deciding what service calls you should block on. Billing the credit card and updating the seat availability are clearly things you need to do now so you can respond with confidence that the booking has been made. Sending the confirmation email isn't time critical, and you don't want to refuse bookings simply because the mail server is down. Therefore, introducing a queue between the mainline business logic execution and the confirmation email service will allow you to do just that: charge the card, update availability, and send the email confirmation when you can.

Introducing a queue on the `emailConfirmationRequests` channel allows the thread passing in the initial message to return as soon as the credit card has been charged and the seat availability has been updated. Changing the Spring Integration configuration to do this is as trivial as adding a child `<queue/>` element to the `<channel/>`:[1]

```
<channel id="bookingConfirmationRequests"/>

<service-activator input-channel="bookingConfirmationRequests"
                   output-channel="chargedBookings"
                   ref="billForBookingService" />

<channel id="chargedBookings" />

<service-activator input-channel="chargedBookings"
                   output-channel="emailConfirmationRequests"
                   ref="seatAvailabilityService" />

<channel id="emailConfirmationRequests">
   <queue />
</channel>

<outbound-channel-adapter channel="emailConfirmationRequests"
                          ref="emailConfirmationService" />
```

[1] This will also require either an explicit or default poller configuration for the consuming endpoint connected to the queue channel.

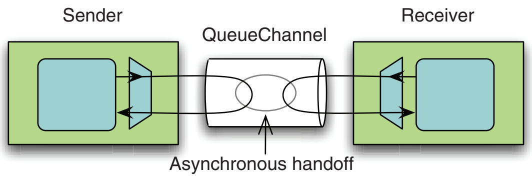


Figure 3 Diagram of threading model of service invocation when using a `QueueChannel`

Let's recap how the threading model changes by introducing the `QueueChannel`, shown in figure 3.

Because a single thread context no longer encompasses all invocations, the transaction boundaries change as well. Essentially, every operation that's executing on a separate thread executes in a separate transaction, as shown in figure 4.

By replacing one of the default channels with a buffered `QueueChannel` and setting up an asynchronous communication model, you gain some confidence that long-running operations won't block the application because some component is down or takes a long time to respond. But now you have another challenge: what if you need to connect one producer with not just one, but two (or more) consumers?

Telling everyone who needs to know that a booking occurred

We've looked at scenarios where a number of services are invoked in sequence with the output of one service becoming the input of the next service in the sequence. This works well when the result of a service invocation needs to be consumed only once, but it's common that more than one consumer may be interested in receiving certain messages. In our current version of the channel configuration, successfully billed bookings that have been recorded by the seat availability service pass directly into a queue for email confirmation. In reality, this information would be of interest to a number of services within the application and systems within the enterprise, such as customer relationship management systems tracking customer purchases to better target promotions and finance systems monitoring the financial health of the enterprise as a whole.

To allow delivery of the same message to more than one consumer, you introduce a publish-subscribe channel after the availability check. The publish-subscribe channel provides one-to-many semantics rather than the one-to-one semantics provided by most channel implementations. One-to-many semantics are particularly useful when you want the flexibility to add additional consumers to the configuration; if the name of the publish-subscribe channel is known, that's all that's required for the configuration of additional consumers with no changes to the core application configuration.

The publish-subscribe channel doesn't support queueing, but it does support asynchronous operation if you provide a task executor that delivers messages to each of

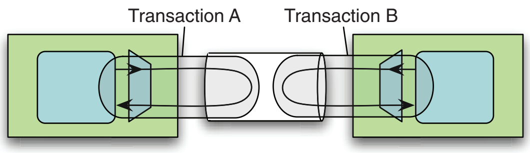


Figure 4 Diagram of transactional boundaries when a `QueueChannel` is used

the subscribers in separate threads. But this approach may still block the main thread sending the message on the channel where the task executor is configured to use the caller thread or block the caller thread when the underlying thread pool is exhausted.

To ensure that a backlog in sending email confirmations doesn't block either the sender thread or the entire thread pool for the task executor, you can connect the new publish-subscribe channel to the existing email confirmation queue by means of a *bridge*. The bridge is an enterprise integration pattern that supports the connection of two channels, which allows the publish-subscribe channel to deliver to the queue and then have the thread return immediately:

```
<channel id="bookingConfirmationRequests"/>

<service-activator input-channel="bookingConfirmationRequests"
                   output-channel="chargedBookings"
                   ref="billForBookingService" />

<channel id="chargedBookings" />

<service-activator input-channel="chargedBookings"
                   output-channel="completedBookings"
                   ref="seatAvailabilityService" />

<publish-subscribe-channel id="completedBookings" />

<bridge input-channel="completedBookings"
        output-channel="emailConfirmationRequests" />

<channel id="emailConfirmationRequests">
    <queue />
</channel>

<outbound-channel-adapter channel="emailConfirmationRequests"
                          ref="emailConfirmationService" />
```

Now it's possible to connect one producer with multiple consumers by means of a publish-subscribe channel. Let's get to the last challenge and emerge victoriously with our dramatically improved application: what if "first come, first served" isn't always right?

Some customers are more equal than others

Let's say you want to ensure that the airline's direct customers have the best possible user experience. To do that, you must prioritize the processing of their requests so you can render the response as quickly as possible. Using a comparator that prioritizes direct customers over indirect, you can modify the first channel to be a priority queue. This causes the framework to instantiate an instance of `PriorityChannel`, which results in a queue that can prioritize the order in which the messages are received. In this case, you provide an instance of a class implementing `Comparator<Message<?>>`:

```
<channel id="bookingConfirmationRequests">
    <priority-queue comparator="customerPriorityComparator" />
</channel>

<service-activator input-channel="bookingConfirmationRequests"
```

```
                  output-channel="chargedBookings"
                  ref="billForBookingService" />

<channel id="chargedBookings" />

<service-activator input-channel="chargedBookings"
                  output-channel="completedBookings"
                  ref="seatAvailabilityService" />

<publish-subscribe-channel id="completedBookings" />

<bridge input-channel="completedBookings"
        output-channel="emailConfirmationRequests" />

<channel id="emailConfirmationRequests">
    <queue />
</channel>

<outbound-channel-adapter channel="emailConfirmationRequests"
                         ref="emailConfirmationService" />
```

The configuration changes made in this section are an example of applying different types of channels for solving the different requirements. Starting with the defaults and working through the example, we replaced several channel definitions with the ones most suitable for each particular situation encountered. What's most important is that every type of channel has its own justification, and what may be advisable in one use case may not be advisable in another. We illustrated the decision process with the criteria we find most relevant in each case.

Summary

The concepts of messages and channels are essential to the flexibility inherent in applications built on Spring Integration. The ease of swapping channel implementations provides a high degree of flexibility in controlling threading models, thread utilization, and latency. In choosing the correct channel, it's vital to understand the behavior of the provided implementations because choosing incorrectly can have serious performance implications or can invalidate the correctness of the application by altering the transactional boundaries. You've learned what channels are, and we gave you examples to help you choose the right channel for the job.

Processing batch jobs

From *Spring Batch in Action*
by Arnaud Cogoluegnes, Thierry Templier,
Gary Gregory, Olivier Bazoud

Another powerful API, Spring Batch, lets developers process data in bulk to and from a variety of data sources, from flat files to databases and anything in between. This excerpt details the components of a Spring Batch job.

The job is the central concept in a batch application: it's the batch process itself. A job has two aspects that we examine in this excerpt: a static aspect used for job modeling and a dynamic aspect used for runtime job management. Spring Batch provides a well-defined model for jobs and includes tools—such as Spring Batch XML—to configure this model. Spring Batch also provides a strong runtime foundation to execute and dynamically manage jobs. This foundation provides a reliable way to control which instance of a job Spring Batch executes and the ability to restart a job where it failed. This section explains these two job aspects: static modeling and dynamic runtime.

Modeling jobs with steps

A Spring Batch job is a sequence of steps configured in Spring Batch XML. Let's delve into these concepts and see what they bring to your batch applications.

Modeling a job

The import products job consists of two steps: decompress the incoming archive and import the records from the expanded file into the database. We could also add a cleanup step to delete the expanded file. Figure 1 depicts this job and its three successive steps.

Decomposing a job into steps is cleaner from both a modeling and a pragmatic perspective because steps are easier to test and maintain than is one monolithic job.

Jobs can also reuse steps; for example, you can reuse the decompress step from the import products job in any job that needs to decompress an archive—you only need to change the configuration.

Figure 1 shows a job built of three successive linear steps, but the sequence of steps doesn't have to be linear, as in figure 2, which shows a more advanced version of the import products job. This version generates and sends a report to an administrator if the read-write step skipped records.

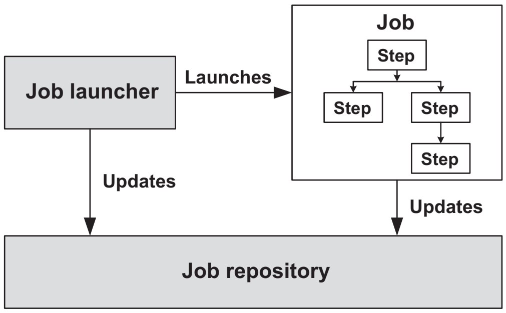


Figure 1 The main Spring Batch components. The framework provides a job repository to store job metadata and a job launcher to launch jobs, and the application developer configures and implements jobs. The infrastructure components—provided by Spring Batch—are in gray, and application components—implemented by the developer—are in white.

To decide which path a job takes, Spring Batch allows for control flow decisions based on the status of the previous step (completed, failed) or based on custom logic (by checking the content of a database table, for example). You can then create jobs with complex control flows that react appropriately to any kind of condition (missing files, skipped records, and so on). Control flow brings flexibility and robustness to your jobs because you can choose the level of control complexity that best suits any given job.

The unpleasant alternative would be to split a big, monolithic job into a set of smaller jobs and try to orchestrate them with a scheduler using exit codes, files, or some other means.

You also benefit from a clear separation of concerns between processing (implemented in steps) and execution flow, configured declaratively or implemented in

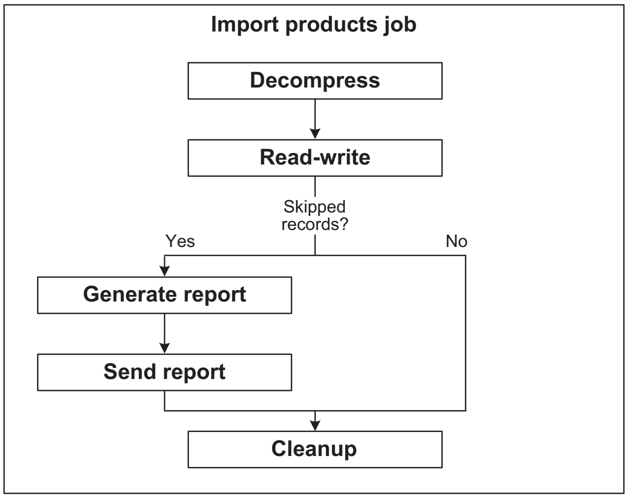


Figure 2 A Spring Batch job can be a nonlinear sequence of steps, like this version of the import products job, which sends a report if some records were skipped.

dedicated decision components. You have less temptation to implement transition logic in steps and thus tightly couple steps with each other.

Let's see some job configuration examples.

Configuring a job

Spring Batch provides an XML vocabulary to configure steps within a job. The following listing shows the code for the linear version of the import products job.

Listing 1 Configuring a job with linear flow

```
<job id="importProductsJob">
  <step id="decompress" next="readWrite">
    <tasklet ref="decompressTasklet" />
  </step>
  <step id="readWrite" next="clean">
    <tasklet>
      <chunk reader="reader" writer="writer"
        commit-interval="100" />
    </tasklet>
  </step>
  <step id="clean">
    <tasklet ref="cleanTasklet" />
  </step>
</job>
```

The `next` attribute of the `step` tag sets the execution flow, by pointing to the next step to execute. Tags like `tasklet` or `chunk` can refer to Spring beans with appropriate attributes.

When a job is made of a linear sequence of steps, using the `next` attribute of the `step` elements is enough to connect the job steps. The next listing shows the configuration for the nonlinear version of the import products job from figure 2.

Listing 2 Configuring a job with nonlinear flow

```
<job id="importProductsJob"
     xmlns="http://www.springframework.org/schema/batch">
  <step id="decompress" next="readWrite">
    <tasklet ref="decompressTasklet" />
  </step>
  <step id="readWrite" next="skippedDecision">      <— Refers to flow decision logic
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="100" />
    </tasklet>
  </step>
  <decision id="skippedDecision"
            decider="skippedDecider">
    <next on="SKIPPED" to="generateReport"/>                  Defines decision
    <next on="*" to="clean" />                                logic
  </decision>
  <step id="generateReport" next="sendReport">
    <tasklet ref="generateReportTasklet" />
  </step>
  <step id="sendReport" next="clean">
    <tasklet ref="sendReportTasklet" />
  </step>
  <step id="clean">
    <tasklet ref="cleanTasklet" />
  </step>
</job>

<bean id="skippedDecider"
      class="com.manning.sbia.ch02.structure.
   ➥  SkippedDecider" />
```

Notice from the previous XML fragment that Spring Batch XML is expressive enough to allow job configuration to be human readable. If your editor supports XML, you also benefit from code completion and code validation when editing your XML job configuration. An integrated development environment like the Eclipse-based Spring-Source Tool Suite also provides a graphical view of a job configuration, as shown in figure 3. To get this graph, open the corresponding XML file and select the Batch-Graph tab at the bottom of the editor.

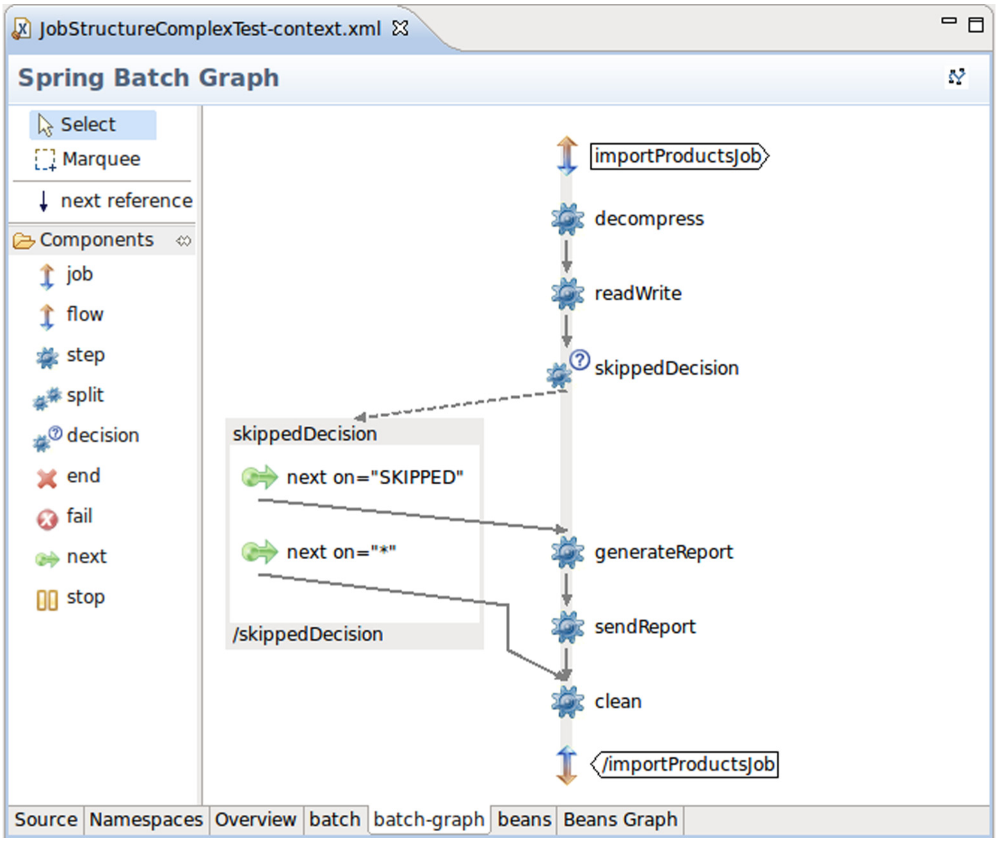


Figure 3 A job flow in the SpringSource Tool Suite. The tool displays a graph based on the job model defined in Spring Batch XML.

NOTE The SpringSource Tool Suite is a free Eclipse-based product that provides tooling for Spring applications (code completion for Spring XML files, bean graphs, and much more). It also provides support for projects in the Spring portfolio like Spring Batch.

Now that you know that a Spring Batch job is a sequence of steps and that you can control job flow, let's see what makes up a step.

Processing with TaskletStep

Spring Batch defines the `Step` interface to embody the concept of a step and provides implementations like `FlowStep`, `JobStep`, `PartitionStep`, and `TaskletStep`. The only implementation you care about as an application developer is `TaskletStep`, which delegates processing to a `Tasklet` object. The `Tasklet` Java interface contains only one method, `execute`, to process some unit of work. Creating a step consists of either writing a `Tasklet` implementation or using one provided by Spring Batch.

You implement your own `Tasklet` when you need to perform processing, such as decompressing files, calling a stored procedure, or deleting temporary files at the end of a job.

If your step follows the classic read-process-write batch pattern, use the Spring Batch XML `chunk` element to configure it as a chunk-processing step. The `chunk` element allows your step to use chunks to efficiently read, process, and write data.

NOTE The Spring Batch `chunk` element is mapped to a `Tasklet` implemented by the `ChunkOrientedTasklet` class.

You now know that a job is a sequence of steps and that you can easily define this sequence in Spring Batch XML. You implement steps with `Tasklets`, which are either chunk oriented or completely customized. Let's move on to the runtime.

Running job instances and job executions

Because batch processes handle data automatically, being able to monitor what they're doing or what they've done is a must. When something goes wrong, you need to decide whether to restart a job from the beginning or from where it failed. To do this, you need to strictly define the identity of a job run and reliably store everything the job does during its run. This is a difficult task, but Spring Batch handles it all for you.

The job, job instance, and job execution

We defined a *job* as a batch process composed of a sequence of steps. Spring Batch also includes the concepts of *job instance* and *job execution*, both related to the way the framework handles jobs at runtime. Table 1 defines these concepts and provides examples.

Table 1 Definitions for job, job instance, and job execution

Term	Description	Example
Job	A batch process, or sequence of steps	The import products job
Job instance	A specific run of a job	The import products job run on June 27, 2010
Job execution	The execution of a job instance (with success or failure)	The first run of the import products job on June 27, 2010

Figure 4 illustrates the correspondence between a job, its instances, and their executions for two days of executions of the import products job.

Now that we've defined the relationship between job, job instance, and job execution, let's see how to define a job instance in Spring Batch.

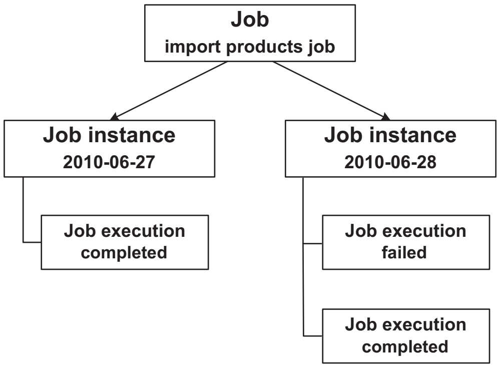


Figure 4 A job can have several job instances, which can have several job executions. The import products job executes daily, so it should have one instance per day and one or more corresponding executions, depending on success or failure.

Defining a job instance

In Spring Batch, a job instance consists of a job and job parameters. When we speak about the June 27, 2010, instance of our import products job, the date is the parameter that defines the job instance (along with the job itself). This is a simple yet powerful way to define a job instance, because you have full control over the job parameters, as shown in the following snippet:

```
jobLauncher.run(job, new JobParametersBuilder()
  .addString("date", "2010-06-27")
  .toJobParameters()
);
```

As a Spring Batch developer, you must keep in mind how to uniquely define a job instance.

JOB INSTANCE A job instance consists of a job and job parameters. We define this contract with the following equation: `JobInstance = Job + JobParameters`.

The previous equation is important to remember. In our example, a job instance is temporal, as it refers to the day it was launched. But you're free to choose what parameters constitute your job instances thanks to job parameters: date, time, input files, or simple sequence counter.

What happens if you try to run the same job several times with the same parameters? It depends on the lifecycle of job instances and job executions.

The lifecycle of a job instance and job execution

Several rules apply to the lifecycle of a job instance and job execution:

- When you launch a job for the first time, Spring Batch creates the corresponding job instance and a first job execution.
- You can't launch the execution of a job instance if a previous execution of the same instance has already completed successfully.
- You can't launch multiple executions of the same instance at the same time.

We hope that by now all these concepts are clear. As an illustration, let's perform runs of the import products job and analyze the job metadata that Spring Batch stores in the database.

Multiple runs of the import products job

To see how Spring Batch updates the job metadata in the persistent job repository previously configured, make the following sequence of runs:

- Run the job for June 27, 2010. The run will succeed.
- Run the job a second time for June 27, 2010. Spring Batch shouldn't launch the job again because it's already completed for this date.
- Run the job for June 28, 2010, with a corrupted archive. The run will fail.
- Run the job for June 28, 2010, with a valid archive. The run will succeed.

Starting the database:

Step 1 Launch the `LaunchDatabaseAndConsole` program.

Running the job for June 27, 2010:

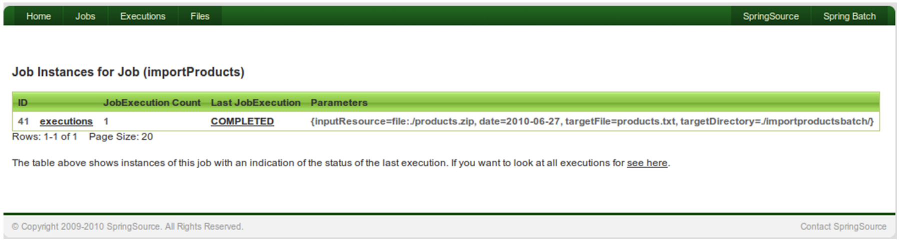
Step 1 Copy the products.zip file from the input directory into the root directory of the ch02 project.

Step 2 Run the `LaunchImportProductsJob` class: this launches the job for June 27, 2010.

Step 3 Run the `LaunchSpringBatchAdmin` program from the code samples to start an embedded web container with the Spring Batch Admin application running.

Step 4 View instances of the import products job at the following URL: http://localhost:8080/springbatchadmin/jobs/importProducts. Figure 5 shows the graphical interface with the job instances and the job repository created for this run.

Step 5 Follow the links from the Job Instances view to get to the details of the corresponding execution, as shown in figure 6.



Job Instances for Job (importProducts)

ID		JobExecution Count	Last JobExecution	Parameters
41	executions	1	COMPLETED	{inputResource=file:./products.zip, date=2010-06-27, targetFile=products.txt, targetDirectory=./importproductsbatch/}

Rows: 1-1 of 1 Page Size: 20

The table above shows instances of this job with an indication of the status of the last execution. If you want to look at all executions for see here.

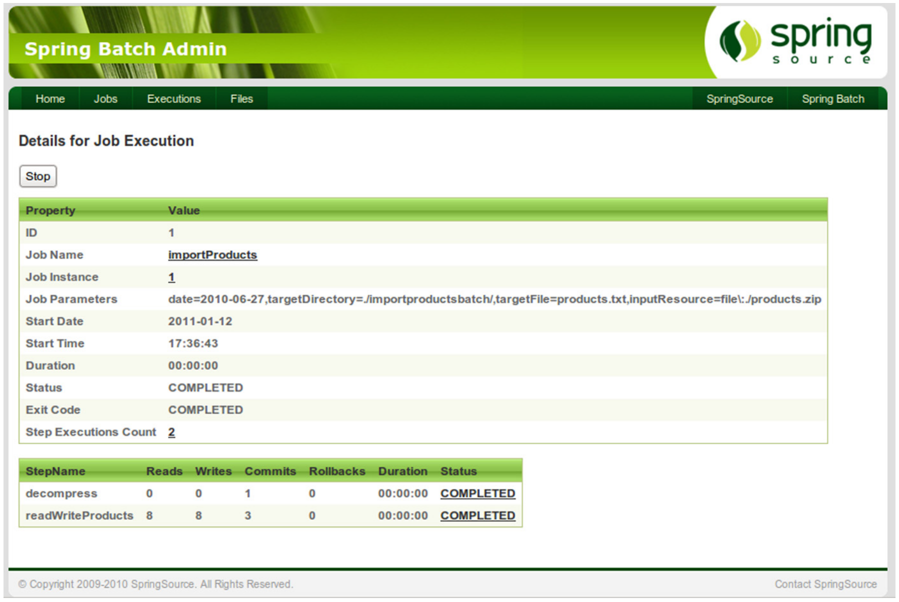
© Copyright 2009-2010 SpringSource. All Rights Reserved. Contact SpringSource

Figure 5 After the run for June 27, 2010, Spring Batch created a job instance in the job repository. The instance is marked as COMPLETED and is the first and only execution to complete successfully.

NOTE You must check the job parameters to be sure of the execution identity. For example, the `date` job parameter tells you that this is an execution of the June 27, 2010, instance. The `Start Date` attribute indicates exactly when the job ran.

Running the job a second time for June 27, 2010:

Step 1 Run the `LaunchImportProductsJob` class. You get an exception because an execution already completed successfully, so you can't launch another execution of the same instance.



Details for Job Execution

Property	Value
ID	1
Job Name	importProducts
Job Instance	1
Job Parameters	date=2010-06-27,targetDirectory=./importproductsbatch/,targetFile=products.txt,inputResource=file\:./products.zip
Start Date	2011-01-12
Start Time	17:36:43
Duration	00:00:00
Status	COMPLETED
Exit Code	COMPLETED
Step Executions Count	2

StepName	Reads	Writes	Commits	Rollbacks	Duration	Status
decompress	0	0	1	0	00:00:00	COMPLETED
readWriteProducts	8	8	3	0	00:00:00	COMPLETED

© Copyright 2009-2010 SpringSource. All Rights Reserved. Contact SpringSource

Figure 6 Details (duration, number of steps executed, and so on) of the first and only job execution for June 27, 2010. You can also learn about the job instance because the job parameters appear in the table.

Running the job for June 28, 2010, with a corrupted archive:

Step 1 Delete the products.zip file and the importproductsbatch directory created to decompress the archive.

Step 2 Copy the products_corrupted.zip from the input directory into the root of the project and rename it products.zip.

Step 3 Simulate launching the job for June 28, 2010, by changing the job parameters in `LaunchImportProductsJob`; for example:

```
jobLauncher.run(job, new JobParametersBuilder()
  .addString("inputResource", "file:./products.zip")
  .addString("targetDirectory", "./importproductsbatch/")
  .addString("targetFile","products.txt")
  .addString("date", "2010-06-28")
  .toJobParameters()
);
```

Step 4 Run the `LaunchImportProductsJob` class. You get an exception saying that nothing can be extracted from the archive (the archive is corrupted).

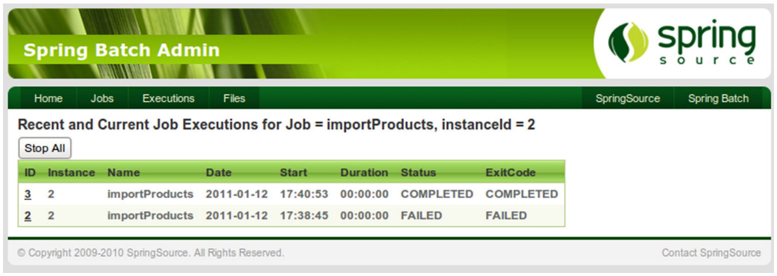
Step 5 Go to http://localhost:8080/springbatchadmin/jobs/importProducts, and you'll see that the import products job has another instance, but this time the execution failed.

Running the job for June 28, 2010 with a valid archive:

Step 1 Replace the corrupted archive with the correct file (the same as for the first run).

Step 2 Launch the job again.

Step 3 Check in Spring Batch Admin that the instance for June 28, 2010, has completed. Figure 7 shows the two executions of the June 28, 2010, instance.



Recent and Current Job Executions for Job = importProducts, instanceId = 2

ID	Instance	Name	Date	Start	Duration	Status	ExitCode
3	2	importProducts	2011-01-12	17:40:53	00:00:00	COMPLETED	COMPLETED
2	2	importProducts	2011-01-12	17:38:45	00:00:00	FAILED	FAILED

© Copyright 2009-2010 SpringSource. All Rights Reserved.

Figure 7 The two June 28, 2010, executions. The first failed because of a corrupted archive, but the second completed successfully, thereby completing the job instance.

NOTE To run the tests from scratch after you run the job several times, stop and restart the `LaunchDatabaseAndConsole` class.

You just put into practice the concepts of job instance and job execution. To do so, you used a persistent job repository, which allowed you to visualize job instances and executions. In this example, job metadata illustrated the concepts, but you can also use this metadata for monitoring a production system.

3.4 Summary

You saw the static and dynamic aspects of jobs: static by modeling and configuring jobs with steps, dynamic through the job runtime handling of job instances and executions. Restarting failed jobs is an important requirement for some batch applications, and you saw how Spring Batch implements this feature by storing job execution metadata; you also saw the possibilities and limitations of this mechanism.

Integrating Spring with JMS

From *Spring Integration in Action*
by Mark Fisher, Jonas Partner,
Marius Bogoevici, and Iwein Fuld

Spring Integration can easily work with traditional JMS. This excerpt explains how to set up a JMS provider and integrate it with the channels to process messages.

For many Java developers, the first thing that comes to mind when they hear "messaging" is the *Java Message Service (JMS)*. That's understandable considering it's the predominant Java-based API for messaging and sits among the standards of the Java Enterprise Edition (JEE). The JMS specification was designed to provide a general abstraction over message-oriented middleware (MOM). Most of the well-known vendor products for messaging can be accessed and used through the JMS API. A number of open source JMS implementations are also available, one of which is *ActiveMQ*, a pure Java implementation of the JMS API. We use ActiveMQ in some of the examples in this excerpt because it's easy to configure as an embedded broker. We don't go into any specific ActiveMQ details, though. If you want to learn more about it, please refer to *ActiveMQ in Action* by Bruce Snyder, Dejan Bosanac, and Rob Davies (Manning, 2011).

Hopefully, by this point in the book, you realize that messaging and event-driven architectures don't necessarily require the use of such systems. In a simple application with no external integration requirements, producer and consumer components may be decoupled by message channels so that they communicate only with messages rather than direct invocation of methods with arguments. Messaging is really a paradigm; the same underlying principles apply whether messaging occurs between components running within the same process or between components running under different processes on disparate systems.

Nevertheless, by supporting JMS, Spring Integration provides a bridge between its simple, lightweight intraprocess messaging and the interprocess messaging that JMS enables across many MOM providers. In this excerpt, you learn how to map between Spring Integration messages and JMS messages. You also learn about several options for integrating with JMS messaging destinations. Spring Integration provides channel adapters and gateways as well as message channel implementations that are backed by JMS destinations. In many cases, the configuration of these elements is straightforward. But, obtaining the most benefit from the available features, such as transactions, requires a thorough understanding of the underlying JMS behavior as dictated by the specification. Therefore, we alternate between the Spring Integration role and the specific JMS details as necessary.

The relationship between Spring Integration and JMS

Spring Integration provides a consistent model for intraprocess and interprocess messaging. The primary role of channel adapters and messaging gateways is to connect a local channel to some external system without impacting the producer or consumer components' code. Another benefit the adapters provide is the separation of the messaging concerns from the underlying transports and protocols. They enable true document-style messaging whether the particular adapter implementation is sending requests over HTTP, interacting with a filesystem, or mapping to another messaging API. The JMS-based channel adapters and messaging gateways fall into that last category and are therefore excellent choices when external system integration is required. Given that the same general messaging paradigm is followed by Spring Integration and JMS, we can conceptualize the intraprocess and interprocess components as belonging to two layers but with a consistent model. See figure 1.

Even though we tend to focus on external system integration when discussing the roles of JMS, there are also benefits to using JMS internally within an application. JMS may be useful between producers and consumers running in the same process because a JMS provider can support persistence, transactions, load balancing, and failover. For this reason, Spring Integration provides a message channel implementation that delegates to JMS behind the scenes. That channel looks like any other channel as far as the

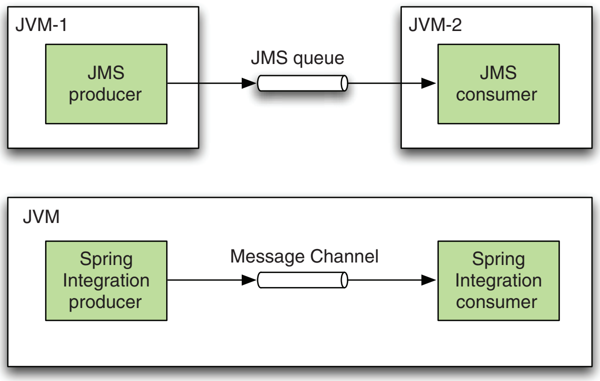


Figure 1 The top configuration shows interprocess integration using JMS. The bottom configuration shows intraprocess integration using Spring Integration. Which type of integration is appropriate depends on the architecture of the application.

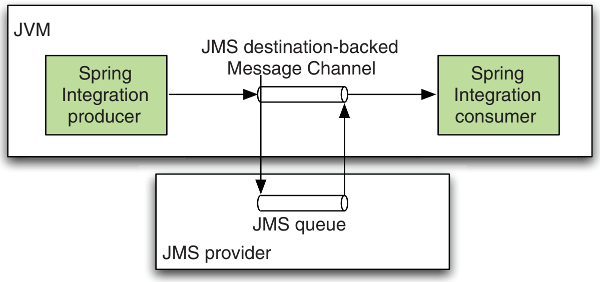

JVM
JMS destination-backed Message Channel
Spring Integration producer
Spring Integration consumer
JMS queue
JMS provider

Figure 2 Design of the destination-backed channel of Spring Integration. It benefits from the guarantees supported by a JMS implementation, but it hides the JMS API behind a channel abstraction.

message-producing and message-consuming components are concerned, so it can be used as an alternative at any point within a message flow as shown in figure 2.

Even for messaging within a single process, the use of a JMS-backed channel provides several benefits. Consider a message channel backed by a simple in-memory queue, as occurs when using the `<queue>` subelement within a `<channel>` without referencing any `MessageStore`. By default, such a channel doesn't *persist* messages to a transactional resource. Instead, the messages are only stored in a volatile queue such that they can be lost in the case of a system failure. They'll even be lost if the process is shut down intentionally before those messages are drained from the queue by a consumer. In certain cases, when dealing with real-time event data that doesn't require persistence, the loss of those event messages upon process termination might not be a problem. It may be well worth the trade-off for asynchronous delivery that allows the producer and consumer to operate within their own threads, on their own schedules. With these message channels backed by a JMS `Destination`, though, you can have the best of both worlds. If persistence and transactions *are* important, but asynchronous delivery is also a desired feature, then these channels offer a good choice even if they're only being used by producers and consumers in the same process space.

The main point here is that even though we often refer to JMS as an option for messaging between a number of individual processes, that's not the *only* time to consider JMS or other interprocess broker-based messaging solutions, such as Advanced Message Queuing Protocol (AMQP), as an option. When multiple processes are involved, the other advantages become more evident. First among these is the natural load balancing that occurs when multiple consuming processes are pulling messages from a shared destination. Unlike producer-side load balancing, the consumers can naturally distribute the load on the basis of their own capabilities. For example, some processes may be running on slower machines or the processing of certain messages may require more resources, but the consumers only ask for more messages when they can handle them rather than forcing some upstream dispatcher to make the decisions.

The second, related benefit is increased scalability. Message-producing processes might be sending more messages than a single consuming process can handle without creating a backlog, resulting in a constantly increasing latency. By adding enough consuming processes to handle the load, the throughput can increase to the point that a backlog no longer exists, or exists only within acceptable limits in rarely achieved high-load situations that occur during bursts of activity from producers.

The third benefit is increased availability. If a consuming process crashes, messages can still be processed as long as one or more other processes are still running. Even if all processes crash, the mediating broker can store messages until those processes come back online. Likewise, on the producing side, processes may come and go without directly affecting any processes on the consuming side. This is nothing more than the benefit of loose coupling inherent in any messaging system, applied not only across producers and consumers themselves but the processes in which they run. Keep in mind when we discuss these scenarios where processes come and go, we're not merely talking about unforeseen system outages. It's increasingly common for modern applications to have "zero downtime" requirements. Such an application must have a distributed architecture with no tight coupling between components in order to accommodate planned downtime of individual processes, one at a time, for system migrations and rolling upgrades.

One last topic we should address briefly here is transactions. In the scenario described previously, where a consuming process crashes or is taken offline while responsible for an in-flight message, transactions play an important role. If the consumer reads a message from a destination but then fails to process it, such as might occur when its host process crashes, then the message might be lost depending on how the system is configured. In JMS, a variety of options correspond to different points on the spectrum of guaranteed delivery. One configuration option is to require an explicit acknowledgment from the consumer. It might be that a consumer acknowledges each message after it successfully stores it on disk. A more robust option is to enable transactions. The consumer would commit the transaction only upon successful processing of the message, and it would roll back the transaction in case of a known failure. When this functionality is enabled, not only do the multiple consuming processes share the load, they can even cover for each other in the event of failures. One consumer may fail while a message is in flight, but its transaction rolls back. The message is then made available to another consuming process rather than being lost.

Table 1 provides a quick overview of the benefits of using JMS with Spring Integration.

It's worth pointing out that the benefits listed in table 1 aren't limited to JMS. Any broker that provides support for reliable messaging across distributed producer and

Table 1 Benefits of using JMS with Spring Integration

Benefit	Description
Load balancing	Multiple consumers in separate virtual machine processes pull messages from a shared destination at a rate determined by their own capabilities.
Scalability	Adding enough consumer processes to avoid a backlog increases throughput and decreases response time.
Availability	With multiple consumer processes, the overall system can remain operational even if one or more individual processes fail. Likewise, consumer processes can be redeployed one at a time to support a rolling upgrade.

consumer processes would provide the same benefits. For example, Spring Integration 2.1 adds support for RabbitMQ, which implements the AMQP protocol. Using the AMQP adapters would offer the same benefits. Likewise, although not as sophisticated, even using Spring Integration's queue-backed channels along with a `MessageStore` can provide the same benefits because that too enables multiple processes to share the work. For now, let's get back to the discussion at hand and explore the mapping of Spring Integration message payloads and headers to and from JMS message instances.

Mapping between JMS and Spring Integration messages

When considering interprocess messaging from the perspective of Spring Integration, the primary role of channel adapters is to handle all of the communication details so that the component on the other side of the message channel has no idea that an external system is involved. That means the channel adapter not only handles the communication via the particular transport and protocol being used, but also must provide a *Messaging Mapper* (http://mng.bz/Fl0P) so that whatever data representation is used by the external system is converted to and from simple Spring Integration messages. Some of that data might map to the *payload* of such a message, whereas other parts of the data might map to the *headers*. That decision should be based on the role of the particular pieces of data, keeping in mind that the headers are typically used by the messaging infrastructure, and the payload is usually the business data that has some meaning within the domain of the application. Thinking of a message as fulfilling the *document message* pattern from Hohpe and Woolf's *Enterprise Integration Patterns* (Addison-Wesley, 2003), the payload represents the document, and the headers contain additional metadata, such as a timestamp or some information about the originating system.

It so happens that the construction of a JMS message, according to the JMS specification, is similar to the construction of a Spring Integration message. This shouldn't surprise you given that the function of the message is the same in both cases. It does mean that the messaging mapper implementation used by the JMS adapters has a simple role. We'll go into the details in a later section, but for now it's sufficient to point out that there are merely some differences in naming. In JMS, the message has a *body*, which is the counterpart of a payload in Spring Integration. Likewise, a JMS message's *properties* correspond to a Spring Integration message's headers. See figure 3.

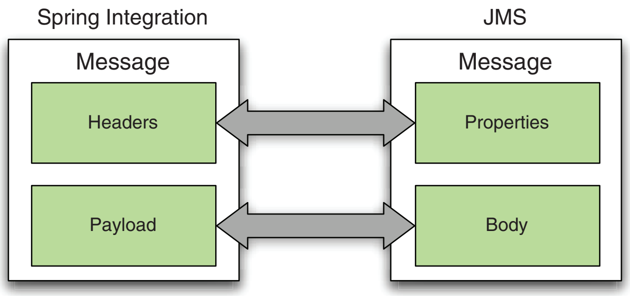


Figure 3 Spring Integration and JMS messages in a side-by-side comparison. The terminology is different, but the structure is the same.

Comparing JMS destinations and Spring Integration message channels

By now you're familiar with the various message channel types available in Spring Integration. One of the most important distinctions we covered is the difference between point-to-point channels and publish-subscribe channels. You saw that when it comes to configuration, the default type for a channel element in XML is point-to-point, and the publish-subscribe channel is clearly labeled as such. The JMS specification uses *destination* instead of *message channel*, but it makes a similar distinction. The two types of JMS `Destination` are `Queues` and `Topics`. A JMS `Queue` provides point-to-point semantics, and a `Topic` supports publish-subscribe interaction. When you use a `Queue`, each message is received by a single consumer, but when you use a `Topic`, the same message can be received by multiple consumers. See table 2 for the side-by-side comparison.

Table 2 Comparing enterprise integration patterns (EIP) to JMS

EIP	JMS
Message Channel	Destination
Point-to-point channel	Queue
Publish-subscribe channel	Topic

Now that we've discussed the relationship between Spring Integration and JMS at a high level, we're almost ready to jump into the details of Spring Integration's JMS adapters. First, it's probably a good idea to take a brief detour through the JMS support in the core Spring Framework. For one thing, the Spring Integration support for JMS builds directly on top of Spring Framework components such as the `JmsTemplate` and the `MessageListener` container. Additionally, the general design of Spring Integration messaging endpoints is largely modeled after the Spring JMS support. You should be able to see the similarities as we quickly walk through the main components and some configuration examples in the next section.

JMS support in the Spring Framework

The logical starting point for any discussion of the Spring Framework's JMS support is the `JmsTemplate`. This is a convenience class for interacting with the JMS API at a high level. Those familiar with Spring are probably already aware of other templates, such as the `JdbcTemplate` and the `TransactionTemplate`. These components are all realizations of the Template pattern described in the Gang of Four's *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al., Addison-Wesley, 1994). Each of these Spring-provided templates satisfies the common goal of simplifying usage of a particular API. One quick example should be sufficient to express this idea. First, we look at code that doesn't use the `JmsTemplate` but instead performs all actions directly with the JMS API. Note that even a simple operation such as sending a text-based message involves a considerable amount of boilerplate code. Here's a simple send-and-receive echo example:

```
public class DirectJmsDemo {

  public static void main(String[] args) {
    try {
      ConnectionFactory connectionFactory =
          new ActiveMQConnectionFactory("vm://localhost");
      Connection connection = connectionFactory.createConnection();
      connection.start();
      int autoAck = Session.AUTO_ACKNOWLEDGE;
      Session session = connection.createSession(false, autoAck);
      Destination queue = new ActiveMQQueue("siia.queue.example1");
      MessageProducer producer = session.createProducer(queue);
      MessageConsumer consumer = session.createConsumer(queue);
      Message messageToSend = session.createTextMessage("Hello World");
      producer.send(messageToSend);
      Message receivedMessage = consumer.receive(5000);
      if (!(receivedMessage instanceof TextMessage)) {
        throw new RuntimeException("expected a TextMessage");
      }
      String text = ((TextMessage) receivedMessage).getText();
      System.out.println("received: " + text);
      connection.close();
    }
    catch (JMSException e) {
      throw new RuntimeException("problem occurred in JMS code", e);
    }
  }

}
```

This code is about as simple as it can get when using the JMS API directly. ActiveMQ enables running an embedded broker (as you can see from the `"vm://localhost"` URL provided to the `ConnectionFactory`). Many JMS providers would be configured within the Java Naming and Directory Interface (JNDI) registry, and that would require additional code to look up the `ConnectionFactory` and `Queue`. Now, let's see how the same task may be performed using Spring's `JmsTemplate`:

```
public class JmsTemplateDemo {

  public static void main(String[] args) {
    ConnectionFactory connectionFactory =
        new ActiveMQConnectionFactory("vm://localhost");
    JmsTemplate jmsTemplate = new JmsTemplate(connectionFactory);
    jmsTemplate.setDefaultDestination(new ActiveMQQueue("siia.queue"));
    jmsTemplate.convertAndSend("hello world");
    System.out.println("received: " + jmsTemplate.receiveAndConvert());
  }

}
```

The code is much simpler, and it also provides fewer chances for developer errors. Any `JMSExceptions` are caught and converted into `RuntimeExceptions` in Spring's `JmsException` hierarchy. The JMS resources, such as `Connection` and `Session`, are also acquired and released as appropriate. In fact, if a transaction is active when this send operation is invoked, and some upstream process has already acquired a JMS

Session, this send operation is executed in the same transactional context. If you've ever worked with Spring's transaction management for data access, this concept should be familiar to you. The idea is roughly the same. If one particular operation in the transaction throws an uncaught RuntimeException, all operations that occurred in that same transactional context are rolled back. If all operations are successful, the transaction is committed.

You probably also noticed that the template method invoked is called convertAndSend and that its argument is an instance of java.lang.String. There are also send() methods that accept a JMS Message you've created, but by using the convertAndSend versions, you can rely on the JmsTemplate to construct the Messages. The conversion itself is a pluggable strategy. The JmsTemplate delegates to an instance of the MessageConverter interface, and the default implementation (SimpleMessageConverter) automatically performs the conversions shown in table 3.

Table 3 Default type conversions

Type passed to SimpleMessageConverter	JMS Message type
java.lang.String	TextMessage
byte[]	BytesMessage
java.util.Map	MapMessage
java.io.Serializable	ObjectMessage

The receiveAndConvert method performs symmetrical conversion *from* a JMS Message. When used on the receiving end, the SimpleMessageConverter extracts the JMS message's body and produces a result with the same mappings shown in the table (for example, TextMessage to java.lang.String).

Sometimes the default conversion options aren't a good fit for a particular application. That's why the MessageConverter is a strategy interface that can be configured directly on the JmsTemplate. Spring provides an object-to-XML (OXM) marshalling version of the MessageConverter that supports any of the implementations of Spring's Marshaller and Unmarshaller interfaces within its toMessage() and fromMessage() methods respectively. For example, an application might be responsible for sending and receiving XML-based text messages over a JMS queue, but the application's developers prefer to hide the XML marshalling and unmarshalling logic in the template itself. Spring's MarshallingMessageConverter may be injected into the JmsTemplate, and in turn, that converter can be injected with one of the options supported by Spring OXM, such as Java Architecture for XML Binding (JAXB).

It's also possible to provide a custom implementation of the Marshaller and Unmarshaller interfaces or even a custom implementation of the MessageConverter. For example, you could implement the MessageConverter interface to create a BytesMessage directly from an object using some custom serialization. That same implementation could then use symmetrical deserialization to map back into objects on the receiving side. Likewise, you might implement the `MessageConverter` interface to map directly between objects and text messages where the actual text content is formatted using JavaScript Object Notation (JSON).

In this section, you learned how the `JmsTemplate` can greatly simplify the code required to do some basic messaging when compared with using the JMS API directly. The examples covered both sending and receiving, but the receive operations were synchronous. Before we discuss how Spring Integration can simplify things even further, we cover the support for asynchronous message reception in the Spring Framework, which provides the foundation upon which the most commonly used JMS adapters in Spring Integration are built.

Asynchronous JMS message reception with Spring

The polling consumer and event-driven consumer patterns are both important. Even with simple intraprocess messaging in a Spring Integration–based application, each pattern has a role in accommodating various message channel options and the use cases that arise from those choices. When dealing with external systems, some transports and protocols are limited to the polling consumer pattern. The JMS model enables both polling and event-driven message reception. This section covers the reasons to consider the event-driven approach and how the Spring Framework supports it, ultimately with message-driven plain old Java objects (POJOs) that keep your code simple and unaware of the JMS API.

Why go asynchronous?

Receiving messages is usually more complicated than sending them. Even though the receiving part of the `JmsTemplate` example looks as simple as the sending part, it's important to recognize that, in that example, the receive operation is synchronous. The `JmsTemplate` has a `receiveTimeout` property. The `JmsTemplate` receive operations return as soon as a message is available at the JMS destination or the timeout elapses, whichever occurs first. That means that if no message is available immediately, the operations may block for as long as indicated by the `receiveTimeout` value.

When relying on blocking receive operations, such `JmsTemplate` usage is an example of the polling consumer pattern. In an application in which an extremely low volume of messages is expected, polling in a dedicated background thread might be okay. But most applications using messaging would prefer to have event-driven consumers.

Support for event-driven consumers could be implemented on top of the simple polling receive calls, though all but the most naive implementations would quickly become complex. A proper and efficient solution requires support for concurrent processing of the received messages. Such a solution would also support transactions, and ideally, it would accommodate a thread pool that adjusts dynamically according to the volume of messages being received. Those same requirements apply to any attempt to adapt an inherently polling-based source of data to an event-driven one. Obviously, that's a common concern for many components in Spring Integration.

As far as the JMS adapters are concerned, the crux of the problem is that the invocation of the JMS receive operation must be performed within the context of the transaction. Then, if the subsequent handling of the message causes a failure, that's most likely a reason to roll back the transaction. Some other consumer may be able to process the message, or perhaps this same handler could handle the message successfully if retried after a brief delay. For example, some system that it relies on might be down at the moment but will be available again shortly. If a JMS consumer rolls back a transaction, then the message won't be removed from the destination; that's what enables redelivery. But if the `Exception` is thrown by the handler in a different thread, it's too late: the JMS consumer has already performed its role, and by passing off the responsibility to an executor that invokes the handler in a different thread, it would've returned successfully after that handoff. It would be unable to react to a rollback based on something that happens later, downstream in the handler's processing of the message content.

Spring's MessageListener container

You're probably convinced that implementing an asynchronous message-driven solution isn't trivial. It's the type of generic, foundational code that should be provided by a framework so developers don't have to spend time dealing with the low-level threading and transaction management concerns. Spring provides this support for JMS with its `MessageListener` containers. Let's look at a modified version of the earlier Hello World example. We use `JmsTemplate` for the sending side only. The message is received by a `MessageListener` asynchronously, and the `DefaultMessageListenerContainer` handles all of the low-level concerns:

```
public class MessageListenerContainerDemo {

  public static void main(String[] args) {

    // establish common resources
    ConnectionFactory connectionFactory =
        new ActiveMQConnectionFactory("vm://localhost");
    Destination queue = new ActiveMQQueue("siia.queue");

    // setup and start listener container
    DefaultMessageListenerContainer container =
        new DefaultMessageListenerContainer();
    container.setConnectionFactory(connectionFactory);
    container.setDestination(queue);
    container.setMessageListener(new MessageListener() {
      public void onMessage(Message message) {
        try {
          if (!(message instanceof TextMessage)) {
            throw new IllegalArgumentException("expected TextMessage");
          }
          System.out.println("received: " +
              ((TextMessage) message).getText());
        }
        catch (JMSException e) {
          throw new RuntimeException(e);
        }
      }
    });
    container.afterPropertiesSet();
    container.start();

    // send Message
    JmsTemplate jmsTemplate = new JmsTemplate(connectionFactory);
    jmsTemplate.setDefaultDestination(queue);
    jmsTemplate.convertAndSend("Hello World");
  }

}
```

The code shows how you can take advantage of asynchronous message reception by depending on Spring's `DefaultMessageListenerContainer` to handle the low-level concerns. Nevertheless, you might be thinking we added a lot more code to the example, and we're back to dealing with some JMS API code directly. For example, we provided an implementation of the JMS `MessageListener` interface, and we're catching the `JMSExceptions` to convert into `RuntimeExceptions` ourselves. In the next section, we take things two steps further. First, we rely on Spring's `MessageListenerAdapter` to decouple our code from the JMS API completely. Second, we refactor the example to use declarative configuration and a dedicated Spring XML schema. In other words, we demonstrate a message-driven POJO.

Message-driven POJOs with Spring

The code and configuration for asynchronous reception can be much simpler. One goal for simplification should be to reduce the dependency on the JMS API. Rather than having to create an implementation of the `MessageListener` interface, you can rely on Spring's `MessageListenerAdapter` to handle that responsibility. It's straightforward: the adapter implements the `MessageListener` interface, but it invokes operations on a delegate when a message arrives. That instance to which it delegates can be any object. The previous code could be updated with the following replacement for the registration of the listener:

```
container.setMessageListener(new MessageListenerAdapter(somePojo));
```

An even better option is to use configuration rather than code. Spring provides a `jms` namespace that supports the configuration of a container and adapter in a few lines of XML:

```
<jms:listener-container>
    <jms:listener destination="aQueue" ref="aPojo" method="someMethod"/>
</jms:listener-container>
```

Many configuration options are available on both the `listener-container` and `listener` elements, but the preceding example provides a glimpse of the simplest possible working case. The XML schema is well documented if you'd like to explore the other options. Our goal here is to provide a sufficient level of background information so you can appreciate that Spring Integration builds directly on top of the JMS support within the base Spring Framework. At this point, you should have a fairly good understanding of that support. We now turn our focus back to Spring Integration to see how it offers an even higher-level approach.

Sending JMS messages from a Spring Integration application

Now that you've seen the similarities between Spring Integration messages and JMS messages and learned about the core Spring support for JMS, you're well prepared to look at the process of sending JMS messages from a Spring Integration application. As with most adapters in Spring Integration, a unidirectional channel adapter and a request/reply gateway are available. Because it's considerably simpler, we begin the discussion with the unidirectional channel adapter.

Spring Integration's outbound JMS channel adapter is a JMS message publisher encapsulated in an implementation of Spring Integration's `MessageHandler` interface. That means it can be connected to any `MessageChannel` so that any Spring Integration `Messages` sent to that channel are converted into JMS `Messages` and then sent to a JMS `Destination`. The JMS `Destination` may be a `Queue` or a `Topic`, but from the perspective of this adapter implementation, that's a configuration detail.

The main class involved in the one-way outbound JMS adapter is called `JmsSendingMessageHandler`. If you look at its implementation, you'll see that it builds completely on the JMS support of the underlying Spring Framework. The most important responsibilities are handled internally by an instance of Spring's `JmsTemplate`. Most of the code in Spring Integration's adapter handles the various configuration options, of which there are many. As far as most users are concerned, even those configuration options are handled by the XML namespace support. In most cases, only a small subset of those options would be explicitly configured, but there are many options for handling more nuanced usage requirements. We walk through several of these in a moment, but first let's look at a typical configuration for one of these adapters:

```
<int-jms:outbound-channel-adapter channel="toJMS"
    destination-name="samples.queue.fromSI" />
```

That looks simple enough, right? Hopefully so, and if you can rely on the defaults, then that's all you need to configure. It's literally adapting the Spring Integration `toJMS` channel so that it converts the messages into JMS `Messages` and then sends them to the JMS `Queue` named `samples.queue.fromSI`. If you want to use a `Topic` instead of a `Queue`, be sure to provide the `pub-sub-domain` attribute with a value of `true`, as in the following example:

```
<int-jms:outbound-channel-adapter channel="toJMS"
    destination-name="samples.topic.fromSI"
    pub-sub-domain="true" />
```

Sometimes it's not practical to rely on just the name of the JMS destination. In fact, it's common that the `Queues` and `Topics` are administered objects that developers should always access via JNDI lookups. Fortunately, you can rely on the Spring Framework's ability to handle that. Instead of using the `destination-name` attribute, you could provide a `destination` attribute whose name is a reference to another object being managed by Spring. That other object could then be a result of a JNDI lookup. For handling that lookup, Spring provides a `FactoryBean` implementation called the `JndiObjectFactoryBean`. Although it's perfectly acceptable to define that `FactoryBean` instance as a low-level `bean` element, there's XML namespace support for much more concise configuration options, as shown here:

```
<int-jms:outbound-channel-adapter channel="toJMS" destination="fromSI"/>

<jee:jndi-lookup id="fromSI" jndi-name="jms/queue.fromSI"/>
```

The functionality would be exactly the same. The difference is limited to the configuration. Access by name is often sufficient in development and testing environments, but JNDI lookups might be required for a production system. In those cases, you can manage the configuration excerpts appropriately by using `import` elements in the configuration or other similar techniques. The important factor is that you don't need to modify any code to handle those different approaches for resolving JMS destinations.

Fortunately, the configuration of the `ConnectionFactory` and `Destinations` can be shared across both the sending and receiving sides. Likewise, for commonly configured references, such as these destinations, there is consistency between the inbound and outbound adapters. In the next section, we focus on the receiving side. We begin with the inbound channel adapter that serves as a polling consumer.

Receiving JMS messages in a Spring Integration application

When receiving JMS messages in a unidirectional way, there are two options. You can define an inbound channel adapter that acts a polling consumer or one that acts as an event-driven consumer. The polling option is configured with an `<inbound-channel-adapter>` element as defined in the JMS schema. It accepts a `destination-name` attribute for the JMS `Queue` or `Topic`. Its default `DestinationResolver` looks up the destination accordingly, and if you need to customize that behavior for some reason, you can provide a `destination-resolver` attribute with the bean name reference of your own implementation. The `<inbound-channel-adapter>` element also requires a poller unless you're relying on a default context-wide poller element. Here's a simple example of an inbound channel adapter that polls for a JMS message every three seconds:

```
<int-jms:inbound-channel-adapter id="pollingJmsInboundAdapter"
    channel="jmsMessages" destination-name="myQueue">
    <int:poller fixed-delay="3000" max-messages-per-poll="1"/>
</int-jms:inbound-channel-adapter>
```

Like the outbound version, if you're specifying a `Topic` name rather than a `Queue` name, you should also provide the `pub-sub-domain` attribute with a value of `true`. If instead you want to reference a Queue or Topic instance, you can use the destination attribute in place of destination-name. This is a common practice when defining this adapter alongside Spring's JNDI support, as shown previously in the outbound channel adapter examples. The following is an example of the corresponding inbound configuration:

```
<int-jms:inbound-channel-adapter id="pollingJmsInboundAdapter"
    channel="jmsMessages" destination="myQueue">
    <int:poller fixed-delay="3000" max-messages-per-poll="1"/>
</int-jms:inbound-channel-adapter>

<jee:jndi-lookup id="myQueue" jndi-name="jms/someQueue"/>
```

As mentioned earlier, polling is rarely the best choice when building a JMS-based solution. Considering that the underlying JMS support in the Spring Framework enables asynchronous invocation of a MessageListener as soon as a JMS message arrives, that's almost always the better option. The only exceptions might be when you want to configure a poller to run infrequently or only at certain times of the day. If the poller is limited to run at certain times of the day, you'd most likely use the cron attribute on a poller element. Other than in those rare situations, the responsiveness will be better and the configuration will be simpler if you stick with Spring Integration's message-driven-channel-adapter. The basic configuration will look the same, but there's no longer a need to define a poller:

```
<int-jms:message-driven-channel-adapter id="messageDrivenAdapter"
    channel="jmsMessages" destination-name="myQueue"/>
```

It may seem odd that, unlike most adapters you've seen, the element doesn't include inbound in its name. Considering it's message-driven, it should be relatively clear that this channel adapter is reacting to inbound JMS messages that arrive at the given Queue or Topic. It sends those messages to the channel referenced by the channel attribute.

Request-reply messaging

The discussion and examples in this excerpt have thus far focused on unidirectional channel adapters. On the sending side, we haven't yet discussed the case where we might be expecting a reply, and on the receiving side, we haven't yet discussed the case where we might be expected to send a reply. We saw that Spring Integration's inbound JMS channel adapters can receive messages with either polling or message-driven behavior. On the other hand, the outbound channel adapter can be used to send messages to a JMS destination, be it a Queue or a Topic. In both cases, the Spring Integration message is mapped to or from the JMS message so that the payload as well as the headers can correspond to the JMS message's body and properties, respectively.

This section introduces Spring Integration's bidirectional gateways for utilizing JMS in a request-reply model of interaction. Much of the functionality, such as mapping between the JMS and Spring Integration message representations, is the same. The difference is that these request-reply gateways are responsible for mapping in both directions. As with the unidirectional channel adapter discussions, we begin with the outbound side. Whereas earlier we could describe the outbound behavior as solely responsible for sending messages, in the gateway case, there is a receive as well, assuming that the JMS reply message arrives as expected. The simplest way to think of the outbound gateway is as a send-and-receive adapter.

The outbound gateway

In the simplest case, the outbound configuration will look similar to the `outbound-channel-adapter` configuration we saw earlier:

```
<int-jms:outbound-gateway request-channel="toJMS"
    reply-channel="jmsReplies"
    request-destination-name="examples.gateway.queue"/>
```

The `request-channel` and `reply-channel` attributes refer to Spring Integration `MessageChannel` instances. Any Spring Integration message that's sent to the request channel will be converted into a JMS message and sent to the gateway's request destination (in this context, *destination* always refers to a JMS component, and *channel* is the Spring Integration message channel). In this example, the destination is a queue. If it were a topic, the `request-pub-sub-domain` attribute would need to be provided with a value of `true`. Because the gateway must manage sending and receiving separately, many of its attributes are qualified as affiliated with either the request or the reply. The `reply-channel` is where any JMS reply messages are sent after they're converted to Spring Integration messages.

You may have noticed that we didn't provide a `reply-destination-name`. That attribute is optional, but it's common to leave it out. The gateway implementation provides the `JMSReplyTo` property on each request message it sends to JMS. If you don't provide a specific destination for that, then it'll handle creation of a temporary queue for that purpose. This assumes that wherever these JMS messages are being sent, a process is in place to check for the `JMSReplyTo` property so it knows where to return a reply message. We discuss the server-side behavior in the next section when we cover the inbound gateway. For the time being, we discuss this interaction with a hypothetical server side where we assume such behavior is in place. The `JMSReplyTo` property is a standard part of the message contract and is defined in the JMS specification. Therefore, it's commonly supported functionality for server-side JMS implementations that accept request messages from a sender who is also expecting reply messages. You must be sure that you're sending to a destination that's backed by a listener implementation with that behavior. The inbound channel adapter we discussed earlier would *not* be a good choice because, as we emphasized, it's intended for unidirectional behavior only. On the other hand, the inbound gateway we discuss in the next section would be a valid option for such server-side request-reply behavior. The core Spring Framework support for message-driven POJOs also supports the `JMSReplyTo` properties of incoming messages as long as the POJO method being invoked has a nonvoid and non-null return value.

The inbound gateway

Like the outbound gateway, Spring Integration's inbound gateway for JMS is an alternative to the inbound channel adapters when request-reply capabilities are required. Perhaps the quickest way to get a sense of what this means is to consider that this adapter covers the functionality we described in abstract terms as the server side in the previous section. The outbound gateway would be the client side as far as that discussion is concerned. The inbound gateway listens for JMS messages, maps each one it receives to a Spring Integration message, and sends that to a message channel. Thus far, that's no different than the role of an inbound channel adapter. The difference is that the message channel in this case would be the initiating end of some pipeline that's expected to produce a reply at some point downstream. When such a reply message is eventually returned to the inbound gateway, it's mapped to a JMS message. The gateway then sends that JMS message to the reply destination. That particular JMS message fulfills the role of the reply message from the client's perspective.

The reply destination is an example of the return address pattern. It may have been provided in the original message's `JMSReplyTo` property, and if so, that takes precedence. If no `JMSReplyTo` property was sent, the inbound gateway falls back to a default reply destination, if configured. As with the request destination, that can either be configured by direct reference or by name. The attribute used for a direct reference is called `default-reply-destination` (again, it's the default because the `JMSReplyTo` property on a request message takes precedence). If configuring the name of the reply destination so that it can be resolved by the gateway's `Destination-Resolver` strategy, use either the `default-reply-queue-name` attribute or the `default-reply-topic-name` attribute. If there's neither a `JMSReplyTo` property on the request message nor a configured reply destination, then an exception will be thrown by the gateway because it would have no way of determining where to send the reply.

That description of the inbound gateway's role probably sounds like it involves a complex implementation. Keep in mind that it builds directly on top of the underlying Spring JMS support that we described earlier. Now you can probably appreciate why we went into considerable detail about that underlying support. As a result, you already have a basic understanding of how the inbound gateway handles the server-side request-reply interaction. As with any Spring Integration inbound gateway, once it maps to a Spring Integration message, it sends that message to a message channel. What makes each gateway unique is *what* it receives and *how* it maps what it receives into a Spring Integration message.

Now that you understand the role of the inbound gateway for JMS, let's look at an example. We start with the simplest configuration options:

```
<int-jms:inbound-gateway id="exampleGateway"
    request-destination-name="someQueue"
    request-channel="requestChannel"/>
```

The `request-channel` indicates where the inbound messages should be sent after they're created by mapping from the JMS source message, and the `request-destination-name` is where those JMS messages are expected to arrive. You may have noticed that no reply-channel attribute is present. This attribute is an optional value for inbound gateways in general. If it's not provided, then the gateway creates a temporary, anonymous channel and sets it as the replyChannel header for the message that it sends downstream. As with the <inbound-channel-adapter> for JMS, the connection-factory attribute is also optional, as long as a JMS ConnectionFactory instance is named connectionFactory in the application context. Add that attribute if for some reason the JMS ConnectionFactory you need to reference has a different bean name.

As you might expect, knowing that we're building on top of Spring's message listener container, a number of other attributes are available. Many of them are passed along to that underlying container. For example, you might want to control the concurrency settings. Following is an example that indicates five core consumers should always be running, but when load increases beyond the capacity of those five, the number of consumers can increase up to 25. At that point, each of those extra consumers can have up to three idle tasks—those where no message is received within the receive timeout of 5 seconds, at which point the consumer will be cleared. The end result of such configuration is that the number of consumers can dynamically fluctuate between 5 and 25 based on the demand for handling incoming messages:

```
<int-jms:inbound-gateway id="exampleGateway"
    request-destination-name="someQueue"
    request-channel="requestChannel"
    concurrent-consumers="5"
    max-concurrent-consumers="25"
    idle-task-execution-limit="3"/>
```

This example shows that various settings of the underlying message listener container can be configured directly on the XML element that represents the Spring Integration gateway. The preceding attributes are a small subset of all the configurable properties of the container. When defining your elements in an IDE with good support for XML, such as the SpringSource Tool Suite, you can easily explore the entire set of available attributes.

We've now covered the various Spring Integration adapters. You saw how such adapters can be used on both the sending side and the receiving side. You saw the unidirectional channel adapters as well as the bidirectional gateways that enable request-reply messaging. Next, we consider the scenario in which the JMS messaging occurs between two applications that are both using Spring Integration.

Messaging between multiple Spring Integration runtimes

In the previous sections, you saw the inbound gateway and the outbound gateway. Both play a role in supporting request-reply messaging, but they were discussed separately thus far. That's because each can be used when you're limited in the assumptions you can make about the application on the other side. As you might expect, the two gateways can work well together when you have Spring Integration applications on both sides. Figure 4 captures this situation

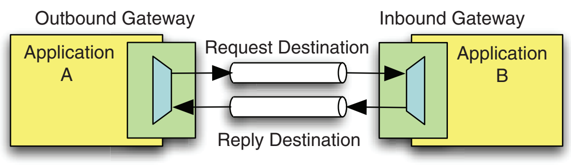


Figure 4 A pair of gateways, one outbound and the other inbound. Each is hosted by a separate Spring Integration application. Those two applications share access to a common JMS broker and a pair of destinations.

In the scenario depicted in the figure, it will obviously be necessary to map between the Spring Integration messages used in each application and the JMS messages that are being passed between the applications. We saw several examples of how the adapters use Spring's JMS `MessageConverter` strategy to convert to and from JMS messages. So far, the examples have mapped between the JMS message body and the Spring Integration message payload. Likewise, the JMS properties have mapped to and from the Spring Integration message headers. These are by far the most common usage patterns for message mapping with JMS, and they make minimal assumptions about the system on the other side of the message exchange.

In a particular deployment environment, though, it might be well known that Spring Integration–based applications exist on both sides of the JMS destination. One Spring Integration application would act as a producer, and the other would act as a consumer. The interaction may be unidirectional, using the channel adapters, or the interaction might involve request-reply exchanges wherein one of the applications contains an outbound gateway and the other contains an inbound gateway (keep in mind that a gateway acts as both a producer and a consumer). If that's the nature of the deployment model, it may or may not be desirable to pass the entire Spring Integration `Message` instance as the JMS message body. The default mapping behavior would obviously work in such an environment, but if you want to "tunnel" through JMS instead, for some reason, then you can override the default configuration. To send a Spring Integration `Message` as the actual body of a JMS message, provide a value of `false` to the `extract-request-payload` property of an outbound gateway:

```
<int-jms:outbound-gateway id="exampleGateway"
    request-destination-name="someQueue"
    request-channel="requestChannel"
    extract-request-payload="false"/>
```

When passing the Spring Integration `Message` as the JMS message body, it's necessary to have a serialization strategy. The standard Java serialization mechanism is one option, because Spring Integration `Message`s implement the `Serializable` interface. One thing to keep in mind when choosing that option is that nonserializable values that are stored in the message headers won't be passed along because they can't be serialized with that approach. An even more important factor to keep in mind is that Java serialization requires that the same class be defined on both the producer and the consumer sides. Not only must it be the same class, but the version of the class must be the same. JMS facilitates this option by providing support for `ObjectMessage`s, where the body is a `Serializable` instance.

At first, it seems convenient to pass your domain objects around without any need to think about conversion or serialization, but it's almost always a bad idea in reality. By requiring exactly the same classes to be available to both the producer and the consumer, this approach violates the primary goal of messaging: loose coupling. Even if you control both sides of the messaging exchange, the fewer assumptions one side makes about the other, the more flexible the application will be. As any experienced developer knows, some of the most regrettable assumptions are those made about the future of an application. For example, if an application needs to evolve to support multiple versions of a certain payload type, reliance on default serialization to and from a single version of a class will be a sure source of regret.

With these twin goals of reducing assumptions and increasing flexibility in mind, let's consider some other options for serializing data. Probably the most common approach in enterprise integration is to rely on XML representations. Spring Integration provides full support for that option with the object-to-XML marshaller and XML-to-object unmarshaller implementations from the Spring Framework's `oxm` module. Another increasingly popular option for serializing and deserializing the payload is to map to and from a portable JSON representation. The advantage of building a solution based on either XML or JSON instead of Java serialization is that the system can be much more flexible. It's not necessary to have the same version on both sides. In fact, as long as the marshaller and unmarshaller implementations account for it, the producer and consumer sides may even convert to and from completely different object types.

Regardless of the chosen serialization mechanism, you must configure a `MessageConverter` on the gateway any time you don't want to rely on the default, which uses Java serialization. You might choose the `MarshallingMessageConverter` provided by Spring for object-to-XML conversion, or you might implement `MessageConverter` yourself. Either way, define the `MessageConverter` within the same `ApplicationContext`, and then provide the reference on the outbound gateway or channel adapter's configuration:

```
<int-jms:outbound-gateway id="exampleGateway"
    request-destination-name="someQueue"
    request-channel="requestChannel"
    extract-request-payload="false"
    message-converter="customConverter"/>

<bean id="customConverter" class="example.CustomMessageConverter"/>
```

Generally, we recommend avoiding the tunneling approach because having the mapping behavior on both sides promotes loose coupling. Even then, it's worth considering the serialization strategy. If a Spring Integration payload is a simple string or byte array, then it'll map to a JMS `TextMessage` or `BytesMessage` respectively when relying on the default `MessageConverter` implementation. The default conversion strategy also provides symmetric behavior when mapping from JMS. A `TextMessage` maps to a string payload, and a `BytesMessage` maps to a byte array payload. But if your Spring Integration payload or JMS body is a domain object, then it's definitely important to consider the degree of coupling because the default `MessageConverter` will rely on Java serialization at that point.

Spring Integration provides bidirectional XML transformers in its XML module and it provides bidirectional JSON transformers in the core module. Both the XML and JSON transformers can be configured using simple namespace-defined elements in XML. You can provide the object-to-XML or object-to-JSON transformer upstream from an outbound JMS channel adapter or gateway, and you can provide the XML-to-object or JSON-to-object transformer downstream from an inbound JMS channel adapter or gateway. One advantage of relying on the transformer instances is that you can reuse them in multiple messaging flows. For example, you might also be receiving XML or JSON from an inbound file adapter, and you might be sending XML or JSON to an outbound HTTP gateway.

If such opportunities for reuse aren't relevant in your particular application, you may prefer to encapsulate the serialization behavior. You can rely on any implementation of Spring's `MessageConverter` strategy interface. As mentioned earlier, the `MarshallingMessageConverter` is available in the Spring Framework. A similar JSON-based implementation was introduced in Spring 3.1, but if you're using an earlier version, it wouldn't be difficult to implement in the meantime. You can provide any other custom logic or delegate to any other serialization library that you choose. If going down that path, you would define the chosen `MessageConverter` implementation as a bean and then reference it from the channel adapter or gateway's `message-converter` attribute.

Summary

Considering the central role that JMS plays in many enterprise Java applications, we wanted to make sure it was clear where Spring Integration overlaps with JMS and where the two can complement each other. You saw the relationship between the two message structures and how to map between them. You also learned how the underlying Spring Framework provides base functionality that greatly simplifies the use of JMS and how Spring Integration takes that even further with its declarative configuration and higher level of abstraction.

Rapid application development using Grails

From *Grails in Action, Second Edition*
by Glen Smith and Peter Ledbrook

Grails, a Groovy-language-based application platform, is based on Hibernate and Spring. Existing for more than five years, it was purchased by SpringSource shortly before VMware bought SpringSource. Grails has a growing, loyal following of developers, mainly because it makes developing web applications a joy and integrates with existing Spring and Hibernate code easily. This excerpt walks through setting up a sample application.

Great strides have been made in the field of Java-based web application frameworks, but creating a new application with them still seems like a lot of work. Grails' core strength is developing web applications quickly, so we'll jump into writing our first application right away.

We'll expose you to the core parts of Grails by developing a simple Quote of the Day application from scratch. You'll store and query the database, develop business logic, write tests, and even add some AJAX functionality. By the end of this discussion, you'll have a good feel for all the basic parts of Grails.

In order to develop serious Grails applications, you'll need a firm grasp of Groovy—the underlying dynamic language that makes Grails tick. In order to get Grails up and running, you'll need to walk through the installation process shown in figure 1.

First, you'll need to have a JDK installed (version 1.5 or later—run `javac-version` from your command prompt to check which version you have). Most PCs come with Java preinstalled these days, so you may be able to skip this step.

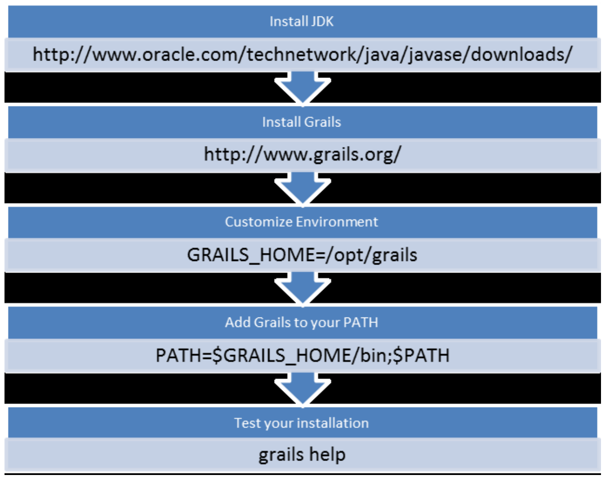


Figure 1 The Grails installation process

Once you're happy that your JDK is installed, download the latest Grails distro from www.grails.org and unzip it to your favorite installation area.

You'll then need to set the `GRAILS_HOME` environment variable, which points to your Grails installation directory, and add GRAILS_HOME/bin to your path. On Mac OS X and Linux, this is normally done by editing the ~/.profile script to contain lines like these:

```
export GRAILS_HOME=/opt/grails
export PATH=$PATH:$GRAILS_HOME/bin
```

On Windows, you'll need to go into System Properties to define `GRAILS_HOME` and update your `PATH` setting.

You can verify that Grails is installed correctly by running `grails help` from the command line. This will give you a handy list of Grails commands, and it'll confirm that everything is running as expected.

When you develop more sophisticated Grails applications, you'll probably want to take advantage of some of the fantastic Grails IDE support out there. There's now Grails plugin support for IntelliJ, NetBeans, and Eclipse—whichever your preferred IDE, there will be a plugin to get you going. We won't be developing too much code, so a basic text editor will be all you need. Fire up your favorite editor, and we'll talk about our sample application.

Our sample program: a Web 2.0 QOTD

If we're going to the trouble of writing a small application, we might as well have some fun. Our example is a quote-of-the-day (QOTD) web application where we'll capture and display famous programming quotes from development rock stars throughout time. We'll let the user add, edit, and cycle through programming quotes, and we'll even add some Ajax sizzle to give it a Web 2.0 feel. We'll want a nice short URL for our application, so let's make "qotd" our application's working title.

NOTE You can download the sample apps for this book, including CSS and associated graphics, from the book's site at manning.com/gsmith.

It's time to get started with our world-changing Web 2.0 quotation app, and all Grails projects begin the same way. First, find a directory to work in. Then create the application:

```
grails create-app qotd
cd qotd
```

Well done. You've created your first Grails application. You'll see that Grails created a qotd subdirectory to hold our application files. Change to that directory now, and we'll stay there for the rest of this discourse.

Because we've done all the hard work of building the application, it'd be a shame not to enjoy the fruit of our labor. Let's give it a run:

```
grails run-app
```

Grails ships with a copy of Jetty (an embeddable Java web server—there is talk that a future version will switch to Tomcat), which Grails uses to host your application during the development and testing lifecycle. When you run the `grails run-app` command, Grails will compile and start your web application. When everything is ready to go, you'll see a message like this on the console:

```
Server running. Browse to http://localhost:8080/qotd
```

This means it's time to fire up your favorite browser and take your application for a spin: http://localhost:8080/qotd/. Figure 2 shows our QOTD application up and running in a browser.

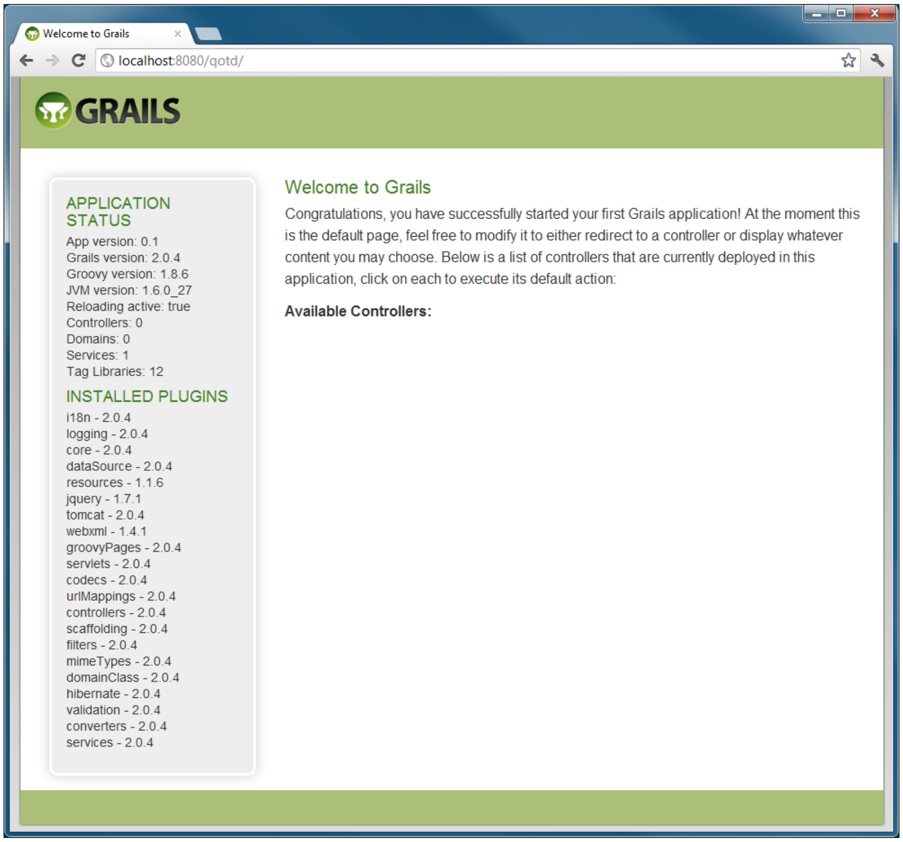
Once you've taken in the home page, you can stop the application by pressing Ctrl-C. Or you can leave it running and issue Grails commands from a separate console window in your operating system.

Running on a custom port (not 8080)

If port 8080 is just not for you (because perhaps you have another process running there, like Tomcat), you can customize the port that the Grails embedded application server runs on using the `-Dserver.port` command-line argument. If you want to run Grails on port 9090, for instance, you could run your application like this:

```
grails -Dserver.port=9090 run-app
```

If you decide to always run a particular application on a custom port, you can create a custom /grails-app/conf/BuildConfig.groovy file with an entry for `grails.server.port.http=9090` to make your custom port the default. Or make a system-wide change by editing the global $HOME/.grails/settings.groovy file.


Welcome to Grails

localhost:8080/qotd/

APPLICATION STATUS

App version: 0.1
Grails version: 2.0.4
Groovy version: 1.8.6
JVM version: 1.6.0_27
Reloading active: true
Controllers: 0
Domains: 0
Services: 1
Tag Libraries: 12

INSTALLED PLUGINS

i18n - 2.0.4
logging - 2.0.4
core - 2.0.4
dataSource - 2.0.4
resources - 1.1.6
jquery - 1.7.1
tomcat - 2.0.4
webxml - 1.4.1
groovyPages - 2.0.4
servlets - 2.0.4
codecs - 2.0.4
urlMappings - 2.0.4
controllers - 2.0.4
scaffolding - 2.0.4
filters - 2.0.4
mimeTypes - 2.0.4
domainClass - 2.0.4
hibernate - 2.0.4
validation - 2.0.4
converters - 2.0.4
services - 2.0.4

Welcome to Grails

Congratulations, you have successfully started your first Grails application! At the moment this is the default page, feel free to modify it to either redirect to a controller or display whatever content you may choose. Below is a list of controllers that are currently deployed in this application, click on each to execute its default action:

Available Controllers:

Figure 2 Our first app is up and running.

Writing your first controller

We have our application built and deployed, but we're a little short on an engaging user experience. Before we go too much further, now's a good time to learn a little about how Grails handles interaction with the user—that's via a *controller*.

Controllers are at the heart of every Grails application. They take input from your user's web browser, interact with your business logic and data model, and route the user to the correct page to display. Without controllers, your web app would be a bunch of static pages.

Like most parts of a Grails application, you can let Grails generate a skeleton controller by using the Grails command line. Let's create a simple controller for handling quotes:

```
grails create-controller quote
```

A Word on Package Naming

If you omit the package name for a Grails artifact, it will default to the name of the app (in the example above if we simply do a “grails create-controller quote”, it will create an artifact called /grails-app/qotd/QuoteController.groovy).

For production code, the Grails community has settled on the standard Java-based convention where your artifacts should be created with your org domain name. Grails lets you change the default package name for your app in /grails-app/conf/Config.groovy. For our featured example, I would change the setting in that file to read:

```
grails.project.groupId = "com.grailsinaction.qotd"
```

With the new setting in play, when I do a “grails create-controller quote” it will create the class in /grails-app/controller/com/grailsinaction/qotd/QuoteController.groovy. It’s a great keysaver change to make at the start of a new Grails project.

Grails will respond by telling you the artifacts it has generated:

```
| Created file grails-app/controllers/qotd/QuoteController.groovy
| Created file grails-app/views/quote
| Created file test/unit/qotd/QuoteControllerTests.groovy
```

Grails will create this skeleton controller in /grails-app/controllers/qotd/QuoteController.groovy. You’ll notice that Grails sorted out the capitalization for you. The basic skeleton is shown in listing 1.

Listing 1 Our first quote controller

```
package qotd

class QuoteController {
    def index() { }
}
```

Not so exciting, is it? The index entry in listing 1 is a Grails *action*, which we’ll return to in a moment. For now, let’s add a home action that sends some text back to the browser—it’s shown in listing 2.

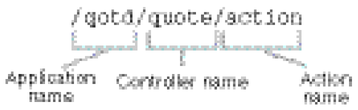
Listing 2 Adding some output

```
package qotd

class QuoteController {
    def index() { }

    def home() {
        render "<h1>Real Programmers do not eat Quiche</h1>"
    }
}
```

Grails provides the `render()` method to send content directly back to the browser. This will become more important when we dip our toes into Ajax waters, but for now let’s use it to deliver our “Real Programmers” heading.

How do we invoke our action in a browser? If this were a Java web application, the URL to get to it would be declared in a configuration file, but not in Grails. This is where we need to introduce you to the Convention over Configuration pattern.

Ruby on Rails introduced the idea that tons of XML configuration (or configuration of any sort) can be avoided if the framework makes some opinionated choices for you about how things will fit together. Grails embraces the same philosophy. Because our controller is called `QuoteController`, Grails will expose its actions over the URL /qotd/quote/youraction. The following gives a visual breakdown of how URLs translate to Grails objects.



In the case of our `hello` action, we'll need to navigate to this URL:

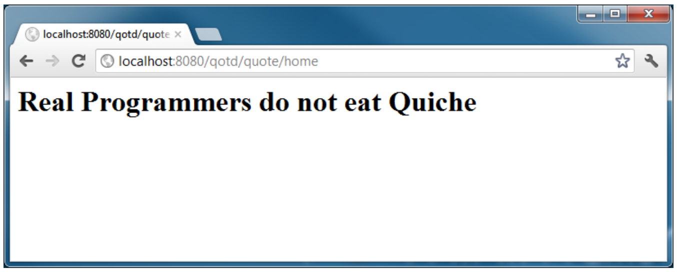
```
http://localhost:8080/qotd/quote/home
```

Figure 3 shows our brand new application up and running, without a single line of XML.

If you were wondering about that `index()` routine in listing 1, that's the method that's called when the user omits the action name. If we decide that all references to /qotd/quote/ should end up at /qotd/quote/home, we need to tell Grails about that with an index action, like the one in listing 3.

Listing 3 Handling redirects

```
package qotd

class QuoteController {
    def index() {
        redirect(action: "home")
    }

    def home() {
        render "<h1>Real Programmers do not eat Quiche</h1>"
    }
}
```



Real Programmers do not eat Quiche

Figure 3 Adding our first bit of functionality

It's looking pretty good so far, but it's pretty nasty to have that HTML embedded in our source. Now that we've learned a little about controllers, it's time to get acquainted with views.

Writing stuff out: the view

Embedding HTML inside your code is always a bad idea. Not only is it difficult to read and maintain, but your graphic designer will need access to your source code in order to design the pages. The solution is to move your display logic out to a separate file, which is known as the *view*, and Grails makes it simple.

If you've done any work with Java web applications, you'll be familiar with JavaServer Pages (JSP). JSPs render HTML to the user of your web application. Grails applications, conversely, make use of Groovy Server Pages (GSP). The concepts are quite similar.

We've already discussed the Convention over Configuration pattern, and views take advantage of the same stylistic mindset. If we create our view files in the right place, everything will hook up without a single line of configuration.

First, in listing 4, we implement our random action. Then we'll worry about the view.

Listing 4 A random quote action

```
def random() {
    def staticAuthor = "Anonymous"
    def staticContent = "Real Programmers don't eat much quiche"
    [ author: staticAuthor, content: staticContent]
}
```

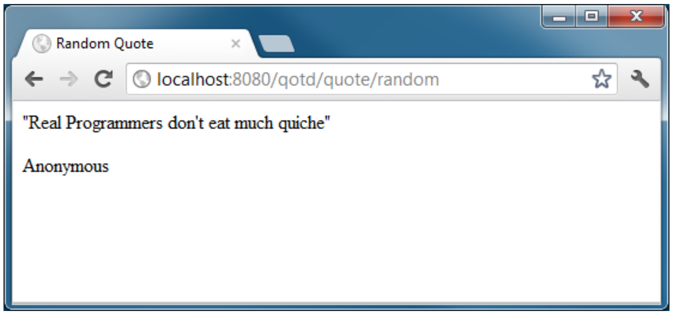
What's all that square bracket-ness? That's how the controller action passes information to the view. If you're an old-school servlet programmer, you might think of it as request-scoped data. The `[:]` operator in Groovy creates a `Map`, so we're passing a series of key/value pairs through to our view.

Where does our view fit into this, and where will we put our GSP file so that Grails knows where to find it? We'll use the naming conventions we used for the controller, coupled with the name of our action, and we'll place our GSP in /grails-app/views/quote/random.gsp. If we follow that pattern, there's no configuration required.

Let's create a GSP file and see how we can reference our `Map` data, as shown in listing 5.

Listing 5 Implementing our first view

```
<html>
<head>
    <title>Random Quote</title>
</head>
<body>
    <q>${content}</q>
    <p>${author}</p>
</body>
</html>
```



"Real Programmers don't eat much quiche"

Anonymous

Figure 4 Our first view in action

The `${content}` and `${author}` format is known as the GSP Expression Language, and if you've ever done any work with JSPs, it will probably be old news to you. If you haven't worked with JSPs before, you can think of those `${}` tags as a way of displaying the contents of a variable. Let's fire up the browser and give it a whirl. Figure 4 shows our new markup in action.

Adding some style with Grails layouts

We now have our first piece of backend functionality written. But the output isn't engaging—there are no gradients, no giant text, no rounded corners. Everything looks pretty mid-90s.

You're probably thinking it's time for some CSS action, but let's plan ahead a little. If we mark up random.gsp with CSS, we're going to have to add those links to the header of every page in the app. There's a better way: Grails layouts.

Layouts give you a way of specifying layout templates for certain parts of your application. For example, we might want all of the quote pages (random, by author, by date) to be styled with a common masthead and navigation links; only the body content should change. To do this, let's first mark up our target page with some IDs that we can use for our CSS. This is shown in listing 6.

Listing 6 Updating the view

```
<html>
<head>
    <title>Random Quote</title>
</head>
<body>
    <div id="quote">
        <q>${content}</q>
        <p>${author}</p>
    </div>
</body>
</html>
```

Now, how can we apply those layout templates (masthead and navigation) we were discussing earlier? Like everything else in Grails, layouts follow a Convention over Configuration style. To have all our `QuoteController` actions share the same layout, we'll create a file called /grails-app/views/layouts/quote.gsp. There are no Grails shortcuts for layout creation, so we've got to roll this one by hand. Listing 7 shows our first attempt at writing a layout.

Listing 7 Adding a layout

```
<html>
    <head>
        <title>QOTD &raquo; <g:layoutTitle/></title>
        <link rel="stylesheet" href="
                <g:createLinkTo dir='css' file='snazzy.css' />
        " />
        <g:layoutHead />
        <r:layoutResources />
    </head>
    <body>
        <div id="header">
            <img src="
                <g:createLinkTo dir='images' file='logo.png'/>
            " alt="logo"/>
        </div>
        <g:layoutBody />
    </body>
</html>
```

1 Merges title from our target page

2 Creates relative link to CSS file

3 Merges head elements from target page

4 Merges in JavaScript, CSS, and other

5 Merges body elements from target page

That's a lot of angle brackets—let's break it down. The key thing to remember is that this is a template page, so the contents of our target page (random.gsp) will be merged with this template before we send any content back to the browser. Under the hood, Grails is using SiteMesh, the popular Java layout engine, to do all of that merging for you. The general process for how SiteMesh does the merge is shown in figure 5.

In order for our layout template in listing 7 to work, it needs a way of accessing elements of the target page (when we merge the title of the target page with the template, for example). That access is achieved through Grails' template taglibs, so it's probably time to introduce you to the notion of taglibs in general.

If you've never seen a tag library (taglib) before, think of it as groups of custom HTML tags that can execute code. In listing 7, we took advantage of the g:create-LinkTo, g:layoutHead, and g:layoutBody tags. When the client's browser requests the page, Grails replaces all of those tag calls with real HTML, and the contents of the HTML will depend on what the individual tag generates. For instance, that first create-LinkTo tag 2 will end up generating a link fragment like /qotd/css/snazzy.css.

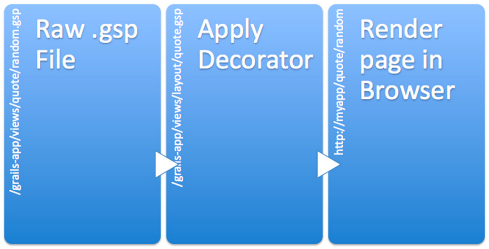

/grails-app/views/quote/random.gsp
Raw .gsp File
/grails-app/views/layout/quote.gsp
Apply Decorator
http://myapp/quote/random
Render page in Browser

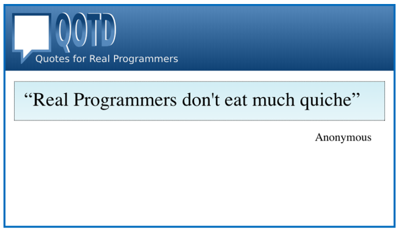
Figure 5 SiteMesh decorates a raw GSP file with a standard set of titles and sidebars

In the title block of the page, we include our QOTD title and then follow it with some chevrons (>>) represented by the HTML character code `&raquo;`, and then add the title of the target page itself ❶.

After the rest of the head tags, we use a `layoutHead` call to merge the contents of the `HEAD` section of any target page ❸. This can be important for search engine optimization (SEO) techniques, where individual target pages might contain their own `META` tags to increase their Google-ability.

Finally, we get to the body of the page. We output our common masthead `div` to get our Web 2.0 gradient and cute icons, and then we call `<g:layoutBody>` to render the `BODY` section of the target page ❹.

Let's refresh our browser to see how we're doing. Figure 6 shows our styled page.



"Real Programmers don't eat much quiche"

Anonymous

Figure 6 QOTD with some funky CSS skinning

Getting the CSS and Artwork

If you're following along step-by-step at your workstation, I'm sure you'll be keen to grab the CSS and image files that go along with the styling above (so your local app can look just the same). You can grab the few files that you need (/web-app/css/snazzy.css and /web-app/images/) directly from the source code that you can download from the book website at manning.com/gsmith.

Our app is looking good. Notice how we've made no changes to our relatively bland random.gsp file. Keeping view pages free of cosmetic markup reduces your maintenance overhead significantly. And if you need to change your masthead, add some more JavaScript includes, or incorporate a few additional CSS files. You do it all in one place: the template.

Fantastic. We're up and running with a controller, view, and template. But things are still pretty static in the data department. We're probably a little overdue to learn how Grails handles stuff in the database. Once we have that under our belt, we can circle back and implement a real random action.

Creating the domain model

We've begun our application, and we can deploy it to our testing web container. But let's not overstate our progress—Google isn't about to buy us just yet. Our app lacks a certain pizzazz. It's time to add some interactivity so that our users can add new quotations to the database. To store those quotations, we're going to need to learn how Grails handles the data model.

Grails uses the term "domain class" to describe those objects that can be persisted to the database. In our QOTD app, we're going to need a few domain classes, but let's start with the absolute minimum: a domain class to hold our quotations.

Let's create a `Quote` domain class:

```
grails create-domain-class quote
```

You'll see that Grails responds by creating a fresh domain class, and a matching unit test to get you started:

```
| Created file grails-app/domain/qotd/Quote.groovy
| Created file test/unit/qotd/QuoteTests.groovy
```

In your Grails application, domain classes always end up under /grails-app/domain. Take a look at the skeleton class Grails has created in /grails-app/domain/qotd/Quote.groovy:

```
package qotd

class Quote {

    static constraints = {
    }
}
```

That's pretty uninspiring. We're going to need some fields in our data model to hold the various elements for each quote. Let's beef up our class to hold the content of the quote, the name of the author, and the date the entry was added, as shown in listing 8.

Listing 8 Our first domain class with teeth

```
package qotd

class Quote {
    String content
    String author
    Date created = new Date()

    static constraints = {
    }

}
```

Now that we've got our data model, we need to go off and create our database schema, right? Wrong. Grails does all that hard work for you behind the scenes. Based on the definitions of the types in listing 8, and by applying some simple conventions, Grails creates a quote table, with `varchar` fields for the strings, and `Date` fields for the date. The next time we run `grails run-app`, our data model will be created on the fly.

But how will it know which database to create the tables in? It's time to configure a data source.

Configuring the data source

Grails ships with an in-memory database out of the box, so if you do nothing, your data will be safe and sound in volatile RAM. The idea of that makes most programmers a little nervous, so let's look at how we can set up a database that's a little more persistent.

In your /grails-app/conf/ directory, you'll find a file named DataSource.groovy. This is where you define the data source (database) that your application will use—you can define different databases for your development, test, and production environments. When you run `grails run-app` to start the local web server, it uses your development data source. Listing 9 shows an extract from the standard DataSource file, which shows the default data source.

Listing 9 Data source definition—in memory

```
development {
    dataSource {
        dbCreate = "create-drop"
        url = " jdbc:h2:mem:devDb;MVCC=TRUE;LOCK_TIMEOUT=10000"
    }
}
```

Re-creates database on every run

Specifies an in-memory database

We have two issues here. The first is that the `dbCreate` strategy tells Grails to drop and re-create your database on each run. This is probably not what you want, so let's change that to `update`, so Grails knows to leave our database table contents alone between runs (but we give it permission to add columns if it needs to).

The second issue relates to the URL—it's using an H2 in-memory database (see www.h2database.com for more info about H2 databases). That's fine for test scripts, but not so good for product development. Let's change it to a file-based version of H2 so we have some real persistence.

Our updated file is shown in listing 10.

Listing 10 Data source definition—persistent

```
development {
    dataSource {
        dbCreate = "update"
        url = " jdbc:h2:devDb;MVCC=TRUE;LOCK_TIMEOUT=10000"
    }
}
```

Preserves tables between runs

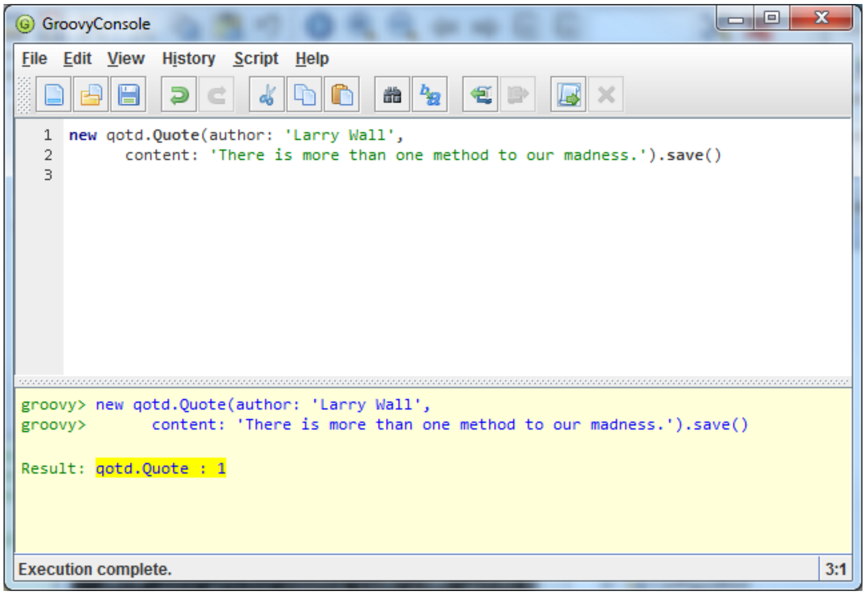
Specifies file-based database

Now we have a database that's persisting our data, so let's look at how we can populate it with some sample data.

Exploring database operations

We haven't done any work on our user interface yet, but it would be great to be able to save and query entries in our quotes table. To do this for now, we'll use the Grails console—a small GUI application that will start your application outside of a web server and give you a console to issue Groovy commands.

You can use the `grails console` command to tinker with your data model before your app is ready to roll. When we issue this command, our QOTD Grails application is bootstrapped, and the console GUI appears, waiting for us to enter some code. Figure 7 shows saving a new quote to the database via the console.



```
new qotd.Quote(author: 'Larry Wall',
      content: 'There is more than one method to our madness.').save()
```

```
groovy> new qotd.Quote(author: 'Larry Wall',
groovy>       content: 'There is more than one method to our madness.').save()

Result: qotd.Quote : 1
```

Execution complete. 3:1

Figure 7 The Grails console lets your run commands from a GUI.

For our first exploration of the data model, it would be nice to create and save some of those `Quote` objects. Type the following into the console window, and then click the Run button (at the far right of the toolbar):

```
new qotd.Quote(author: 'Larry Wall',
      content: 'There is more than one method to our madness.').save()
```

The bottom half of the console will let you know you're on track:

```
Result: qotd.Quote : 1
```

Where did that `save()` routine come from? Grails automatically endows domains with certain methods. Let's add a few more entries, and we'll get a taste of querying:

```
new qotd.Quote(author: 'Chuck Norris Facts', content: 'Chuck Norris always
     uses his own design patterns, and his favorite is the Roundhouse
     Kick').save()
new qotd.Quote(author: 'Eric Raymond', content: 'Being a social outcast helps
     you stay concentrated on the really important things, like thinking and
     hacking.').save()
```

Let's use another one of those dynamic methods (`count()`) to make sure that our data is being saved to the database correctly:

```
println qotd.Quote.count()
3
```

Looks good so far. It's getting a bit tedious typing in that `qotd` package name before every command, so let's put an import into our script so we can cut down on the boilerplate and get on with business:

```
import qotd.*
println Quote.count()
3
```

Much clearer. Next it's time to roll up our sleeves and do some querying on our Quote database. To simplify database searches, Grails introduces special query methods on your domain class called *dynamic finders*. These special methods utilize the names of fields in your domain model to make querying as simple as this:

```
import qotd.*
def quote = Quote.findByAuthor("Larry Wall")
println quote.content
There is more than one method to our madness.
```

Now that we know how to save and query, it's time to start getting our web application up and running. Exit the Grails console, and we'll learn a little about getting those quotes onto the web.

Adding UI actions

Let's get something on the web. First, we'll need an action on our `QuoteController` to return a random quote from our database. We'll work out the random selection later—for now, let's cut some corners and fudge our sample data, as shown in listing 11.

Listing 11 Random refactored

```
def random() {
    def staticQuote = new Quote(author: "Anonymous",
                content: "Real Programmers Don't eat quiche")
    [ quote : staticQuote]
}
```

We'll also need to update our /grails-app/views/quote/random.gsp file to use our new `Quote` object:

```
<q>${quote.content}</q>
<p>${quote.author}</p>
```

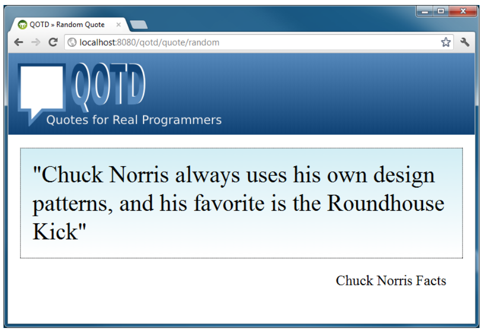
There's nothing new here, just a nicer data model. This would be a good time to refresh your browser and see our static quote being passed through to the view. Give it a try to convince yourself it's all working.

Now that you have a feel for passing model objects to the view, and now that we know enough querying to be dangerous, let's rework our action in listing 12 to implement a real random database query.

Listing 12 A database-driven random query

```
def random() {
    def allQuotes = Quote.list()                                   ◁ 1 Obtains list of quotes
    def randomQuote
    if (allQuotes.size() > 0) {
        def randomIdx = new Random().nextInt(allQuotes.size())     ◁ 2 Selects random quote
        randomQuote = allQuotes[randomIdx]
    } else {
        randomQuote = new Quote(author: "Anonymous",          ③ Generates default quote
            content: "Real Programmers Don't eat Quiche")
    }
    [ quote : randomQuote]                          ④ Passes quote to the view
}
```

With our reworked `random` action, we're starting to take advantage of some real database data. The `list()` method ① will return the complete set of `Quote` objects from the quote table in the database and populate our `allQuotes` collection. If there are any entries in the collection, we select a random one ② based on an index into the collection; otherwise, we use a static quote ③. With all the heavy lifting done, we return a `randomQuote` object to the view in a variable called `quote` ④, which we can access in the GSP file.



"Chuck Norris always uses his own design patterns, and his favorite is the Roundhouse Kick"

Chuck Norris Facts

Figure 8 Our random quote feature in action

Now that we've got our random feature implemented, let's head back to http://localhost:8080/qotd/quote/random to see it in action. Figure 8 shows our random feature in action.

Scaffolding: just add rocket fuel

We've done all the hard work of creating our data model. Now we need to enhance our controller to handle all the CRUD actions to let users put their own quotes in the database.

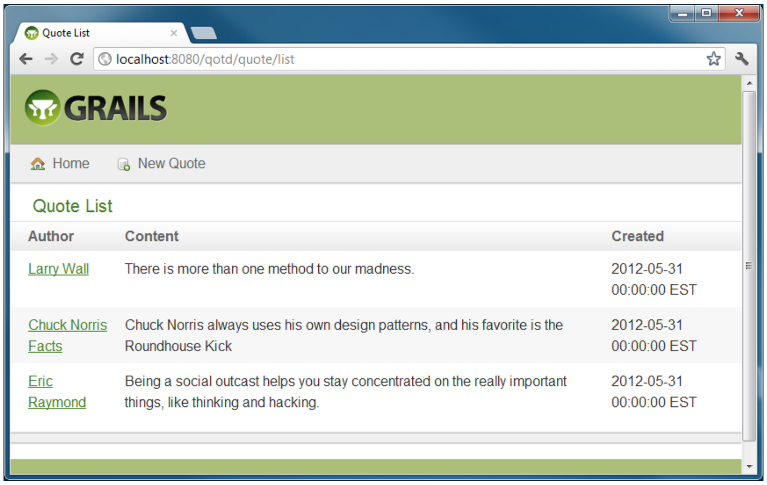
That's if we want to do a slick job of it. But if we want to get up and running quickly, Grails offers us a fantastic shortcut called *scaffolding*. Scaffolds dynamically implement basic controller actions and views for the common things you'll want to do when CRUDing your data model.

How do we scaffold our screens for adding and updating quote-related data? It's a one-liner for the `QuoteController`, as shown in listing 13.

Listing 13 Enabling scaffolding

```
class QuoteController {
     static scaffold = true
    // our other stuff here...
}
```

That's it. When Grails sees a controller marked as `scaffold = true`, it goes off and creates some basic controller actions and GSP views on the fly. If you'd like to see it in action, head over to http://localhost:8080/qotd/quote/list and you'll find something like the edit page shown in figure 9.



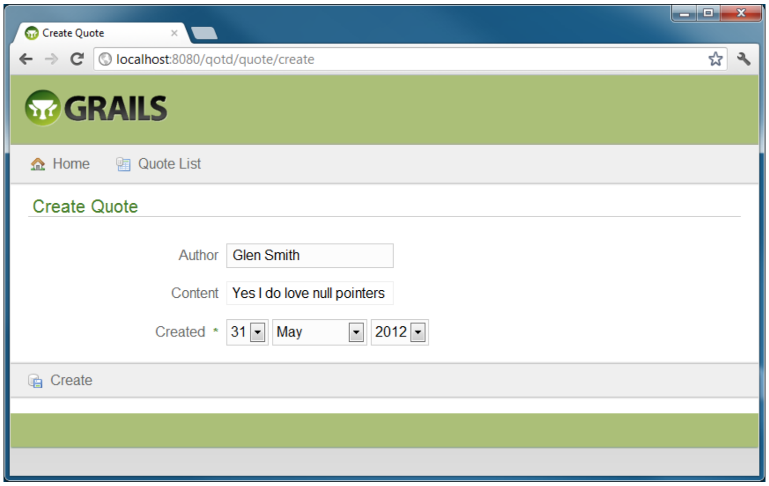
Quote List

Author	Content	Created
Larry Wall	There is more than one method to our madness.	2012-05-31 00:00:00 EST
Chuck Norris Facts	Chuck Norris always uses his own design patterns, and his favorite is the Roundhouse Kick	2012-05-31 00:00:00 EST
Eric Raymond	Being a social outcast helps you stay concentrated on the really important things, like thinking and hacking.	2012-05-31 00:00:00 EST

Figure 9 The `list()` scaffold in action

Click the New Quote button, and you'll be up and running. You can add your new quote as shown in figure 10.

That's a lot of power to get for free. The generated scaffolds are probably not tidy enough for your public-facing sites, but they're absolutely fantastic for your admin screens and perfect for tinkering with your database during development (where you don't want the overhead of mocking together a bunch of CRUD screens).



Create Quote

Author	Glen Smith
Content	Yes I do love null pointers
Created *	31 May 2012

Figure 10 Adding a quote has never been easier.

Surviving the worst case scenario

Our model is looking good and our scaffolds are great, but we're still missing some pieces to make things a little more robust. We don't want users putting dodgy stuff in our database, so let's explore some validation.

Validation is declared in our `Quote` object, so we just need to populate the `constraints` closure with all the rules we'd like to apply. For starters, let's make sure that users always provide a value for the author and content fields, as shown in listing 14.

Listing 14 Adding basic validation

```
class Quote {
    String content
    String author
    Date created = new Date()
    static constraints = {
            author(blank:false)
            content(maxSize:1000, blank:false)
    }
}
```

Enforces data validation

These constraints tell Grails that neither `author` nor `content` can be blank (neither `null` nor `0` length). If we don't specify a size for `String` fields, they'll end up being defined `VARCHAR(255)` in our database. That's probably fine for `author` fields, but our content may expand on that a little. That's why we've added a `maxSize` constraint.

Entries in the `constraints` closure also affect the generated scaffolds. For example, the ordering of entries in the `constraints` closure also affects the order of the fields in generated pages. Fields with constraint sizes greater than 255 characters are rendered as `HTML TEXTAREAs` rather than `TEXT` fields. Figure 11 shows how error messages display when constraints are violated.

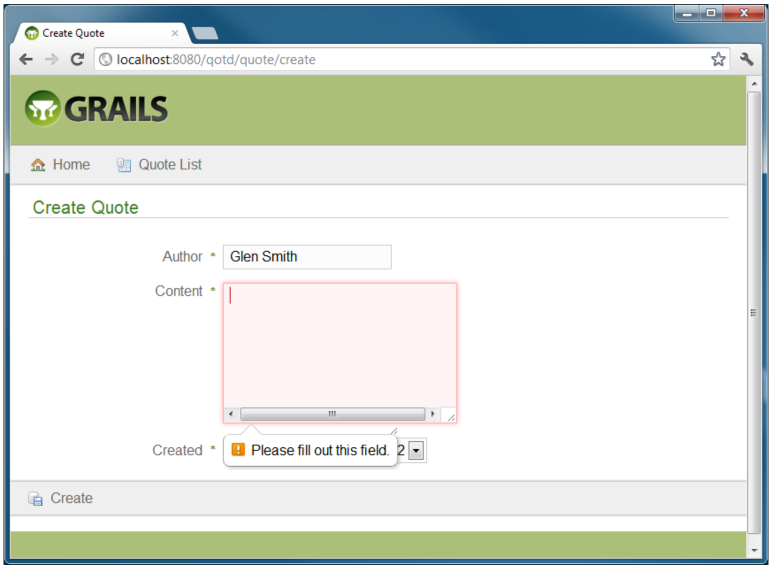


Figure 11 When constraints are violated, error messages appear in red.

Summary and best practices

Congratulations, you've written and deployed your first Grails app, and now you have a feel for working from scratch to completed project. The productivity rush can be quite addictive.

Here are a few key tips you should take-away from this discussion:

- *Rapid iterations are key.* The most important take-away is that Grails fosters rapid iterations to get your application up and running in record time, and you'll have a lot of fun along the way.
- *Noise reduction fosters maintenance and increases velocity.* By embracing Convention over Configuration, Grails gets rid of tons of XML configuration that used to kill Java web frameworks.
- *Bootstrapping saves time.* For the few cases where you do need scaffolding code (for example, in UI design), Grails generates all the shell boilerplate code to get you up and running. This is another way Grails saves you time.