Spring Web Flow Reference Guide

Version 2.5.1.RELEASE

Keith Donald , Erwin Vervaet , Jeremy Grelle , Scott Andrews , Rossen Stoyanchev , Phillip Webb

Table of Contents

Preface

Many web applications require the same sequence of steps to execute in different contexts. Often these sequences are merely components of a larger task the user is trying to accomplish. Such a reusable sequence is called a flow.

Consider a typical shopping cart application. User registration, login, and cart checkout are all examples of flows that can be invoked from several places in this type of application.

Spring Web Flow is the module of Spring for implementing flows. The Web Flow engine plugs into the Spring Web MVC platform and provides declarative flow definition language. This reference guide shows you how to use and extend Spring Web Flow.

1. Introduction

1.1. What this guide covers

This guide covers all aspects of Spring Web Flow. It covers implementing flows in end-user applications and working with the feature set. It also covers extending the framework and the overall architectural model.

1.2. What Web Flow requires to run

Java 1.8 or higher.

Spring 5.0 or higher.

1.3. Resources

You can ask questions and interact on StackOverflow using the designated tags, see Spring at StackOverflow.

Report bugs and make requests using the Spring Issue Tracker.

Submit pull requests and work with the source code , see Web Flow on Github.

1.4. How to access Web Flow artifacts from Maven Central

Each jar in the Web Flow distribution is available in the Maven Central Repository. This allows you to easily integrate Web Flow into your application if you are already using Maven as the build system for your web development project.

To access Web Flow jars from Maven Central, declare the following dependency in your pom:

```
<dependency>
    <groupId>org.springframework.webflow</groupId>
    <artifactId>spring-webflow</artifactId>
    <version>x.y.z.RELEASE</version>
</dependency>
```

If using JavaServer Faces, declare the following dependency in your pom (includes transitive dependencies "spring-binding", "spring-webflow"):

```
<dependency>
    <groupId>org.springframework.webflow</groupId>
    <artifactId>spring-faces</artifactId>
    <version>x.y.z.RELEASE</version>
</dependency>
```

1.5. How to access nightly builds and milestone releases

Nightly snapshots of Web Flow development branches are available using Maven. These snapshot builds are useful for testing out fixes you depend on in advance of the next release, and provide a convenient way for you to provide feedback about whether a fix meets your needs.

Accessing snapshots and milestones with Maven

For milestones and snapshots you'll need to use the SpringSource repository. Add the following repository to your Maven pom.xml:

```
<repository>
    <id>spring</id>
    <name>Spring Repository</name>
    <url>http://repo.spring.io/snapshot</url>
</repository>
```

Then declare the following dependencies:

```
<dependency>
    <groupId>org.springframework.webflow</groupId>
    <artifactId>spring-webflow</artifactId>
    <version>x.y.z.BUILD-SNAPSHOT</version>
</dependency>
```

And if using JSF:

```
<dependency>
    <groupId>org.springframework.webflow</groupId>
    <artifactId>spring-faces</artifactId>
    <version>x.y.z.BUILD-SNAPSHOT</version>
</dependency>
```

2. What's New

2.1. Spring Web Flow 2.5

This release provides an upgrade path to Spring Framework 5 that in turn requires Java 8+, Servlet 3.1, Hibernate 5, Tiles 3. See the Spring Framework wiki for more details. The samples repository has been upgraded to Spring Web Flow 2.5.

As of 2.5 there is no longer a *spring-js* module. The classes from that module have been kept but moved to new packages in the *spring-webflow* module. The *spring-js-resources* module is available as an optional module that must be included explicitly.

This release requires JSF 2.2 or higher.

2.2. Spring Web Flow 2.4

This release requires JDK 1.6.

Java-based Configuration

Web Flow now supports a Java-based alternative for its system configuration. See the updated Chapter 10, *System Setup*.

Also see the booking-mvc and booking-faces samples that have been updated to use all Java config.

Spring MVC Flash Scope Integration

When a flow ends it can now redirect to a Spring MVC controller after saving attributes in Spring MVC's flash scope for the controller to access.

See Section 11.8, “Saving Flow Output to MVC Flash Scope”.

Partial JSR-303 Bean Validation

A flow definition can apply partial validation on the model through the validation-hints attribute supported on view state and transition elements.

See the section called “Partial Validation”.

Hibernate Support

The `HibernateFlowExecutionListener` now supports Hibernate 4 in addition to Hibernate 3.

As of 2.4.4 the `HibernateFlowExecutionListener` also works with Hibernate 5.

Tiles 3 Support

The `AjaxTilesView` now supports Tiles 3 in addition to Tiles 2.2.

Minimum JSF 2.0 Requirement

Java ServerFaces version 1.2 and earlier are no longer supported by Spring Web Flow, if you have not done so already you will need to upgrade to JSF 2.0 or above. In addition the Spring Faces components that were previously provided with JSF 1.2 for progressive AJAX enhancements have been removed in this release.

See ???.

Portlet API 2.0 and JSF 2.0 support

The internal Portlet integration introduced in Spring Web Flow 2.2 has been upgraded for JSF 2.0 compatibility. Some of the more advanced JSF 2.0 features, such as partial state saving, are not supported in a Portlet environment, however, existing application can now upgrade to the minimum required JSF version. Upgraded projects will need to ensure that the `<faces:resources>` elements is included as part of their Spring configuration.

Deprecations

This release deprecates *Spring.js*. The deprecation includes the entire *spring-js-resources* module including *Spring.js* and *Spring-Dojo.js* and the bundled Dojo and CSS Framework. Also deprecated is the `SpringJavascriptAjaxHandler` from the *spring-js* module. The rest of *spring-js*, e.g. `AjaxHandler`, `AjaxTilesView`, will be folded into *spring-webflow* in a future release.

OGNL support is now deprecated.

2.3. Spring Web Flow 2.3

Embedding A Flow On A Page

By default Web Flow does a client-side redirect upon entering every view state. That makes it impossible to embed a flow on a page or within a modal dialog and execute more than one view state without causing a full-page refresh. Web Flow now supports launching a flow in "embedded" mode. In this mode a flow can transition to other view states without a client-side redirect during Ajax requests. See Section 11.7, “Embedding A Flow On A Page” and Section 13.6, “Embedding a Flow On a Page”.

Support For JSR-303 Bean Validation

Support for the JSR-303 Bean Validation API is now available building on equivalent support available in Spring MVC. See Section 5.10, “Validating a model” for more details.

Flow-Managed Persistence Context Propagation

Starting with Web Flow 2.3 a flow managed `PersistenceContext` is automatically extended (propagated) to sub-flows assuming the subflow also has the feature enabled as well. See Section 7.3, “Flow Managed Persistence And Sub-Flows”.

Portlet 2.0 Resource Requests

Support for Portlet 2.0 resource requests has now been added enabling Ajax requests with partial rendering. URLs for such requests can be prepared with the `<portlet:resourceURL>` tag in JSP pages. Server-side processing is similar to a combined an action and a render requests but combined in a single request. Unlike a render request, the response from a resource request includes content from the target portlet only.

Custom ConversationManager

The `<flow-execution-repository>` element now provides a conversation-manager attribute accepting a reference to a ConversationManager instance.

Redirect In Same State

By default Web Flow does a client-side redirect when remaining in the same view state as long as the current request is not an Ajax request. This is useful after form validation failure. Hitting Refresh or Back won't result in browser warnings. Hence this behavior is usually desirable. However a new flow execution attribute makes it possible to disable it and that may also be necessary in some cases specific to JSF applications. See Section 13.7, “Redirect In Same State”.

Samples

The process for building the samples included with the distribution has been simplified. Maven can be used to build all samples in one step. Eclipse settings include source code references to simplify debugging.

Additional samples can be accessed as follows:

```
mkdir spring-samples
cd spring-samples
svn co https://src.springframework.org/svn/spring-samples/webflow-primefaces-showcase
cd webflow-primefaces-showcase
mvn package
# import into Eclipse
```

```
mkdir spring-samples
cd spring-samples
svn co https://src.springframework.org/svn/spring-samples/webflow-showcase
cd webflow-showcase
mvn package
# import into Eclipse
```

2.4. Spring Web Flow 2.2

JSF 2 Support

Comprehensive JSF 2 Support

Building on 2.1, Spring Web Flow version 2.2 adds support for core JSF 2 features The following features that were not supported in 2.1 are now available: partial state saving, JSF 2 resource request, handling, and JSF 2 Ajax requests. At this point support for JSF 2 is considered comprehensive although not covering every JSF 2 feature -- excluded are mostly features that overlap with the core value Web Flow provides such as those relating to navigation and state management.

See Section 13.3, “Configuring Web Flow for use with JSF” for important configuration changes. Note that partial state saving is only supported with Sun Mojarra 2.0.3 or later. It is not yet supported with Apache MyFaces. This is due to the fact MyFaces was not as easy to customize with regards to how component state is stored. We will work with Apache MyFaces to provide this support. In the mean time you will need to use the `javax.faces.PARTIAL_STATE_SAVING` context parameter in `web.xml` to disable partial state saving with Apache MyFaces.

Travel Sample With the PrimeFaces Components

The main Spring Travel sample demonstrating Spring Web Flow and JSF support is now built on JSF 2 and components from the PrimeFaces component library. Please check out the booking-faces sample in the distribution.

Additional samples can be found at the Spring Web Flow - Prime Faces Showcase, an SVN repository within the spring-samples repository. Use these commands to check out and build:

```
svn co https://src.springframework.org/svn/spring-samples/webflow-primefaces-showcase
 cd webflow-primefaces-showcase
 mvn package
```

Spring Security Facelets Tag Library

A new Spring Security tag library is available for use with with JSF 2.0 or with JSF 1.2 Facelets views. It provides an <authorize> tag as well as several EL functions. See Section 13.9, “Using the Spring Security Facelets Tag Library” for more details.

Spring JavaScript Updates

Deprecated ResourcesServlet

Starting with Spring 3.0.4, the Spring Framework includes a replacement for the ResourcesServlet. Please see the Spring Framework documentation for details on the custom mvc namespace, specifically the new "resources" element.

Dojo 1.5 and dojox

The bundled custom Dojo build is upgraded to version 1.5. It now includes dojox.

Note that applications are generally encouraged to prepare their own custom Dojo build for optimized performance depending on what parts of Dojo are commonly used together. For examples see the scripts used by Spring Web Flow to prepare its own custom Dojo build.

Two Spring JS artifacts

The `spring-js` artifact has been split in two -- the new artifact (`spring-js-resources`) contains client side resource (.js, .css, etc.) while the existing artifact (`spring-js`) contains server-side Java code only.

Applications preparing their own custom Dojo build have an option now to avoid including `spring-js-resources` and put `Spring.js` and `Spring-Dojo.js` directly under the root of their web application.

Client resources moved into META-INF/web-resources

Bundled client resources (.js, .css, etc.) have been moved to `META-INF/web-resources` from their previous location under `META-INF`. This change is transparent for applications but will result in simpler and safer configuration when using the new resource handling mechanism available in Spring 3.0.4.

JSF Portlet Support

Portlet API 2.0 and JSF 1.2 support

In previous versions of Spring Web Flow support for JSF Portlets relied on a Portlet Bridge for JSF implementation and was considered experimental. Spring Web Flow 2.2 adds support for JSF Portlets based on its own internal Portlet integration targeting Portlet API 2.0 and JSF 1.2 environments. See ??? for more details. The Spring Web Flow Travel JSF Portlets sample has been successfully tested on the Apache Pluto portal container.

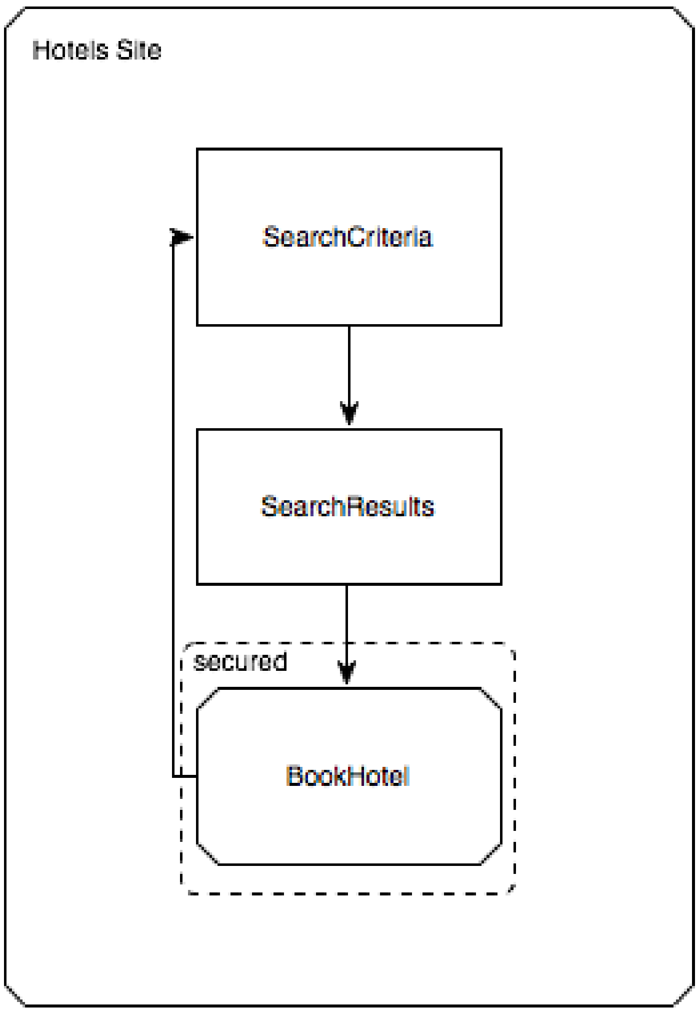
3. Defining Flows

3.1. Introduction

This chapter begins the Users Section. It shows how to implement flows using the flow definition language. By the end of this chapter you should have a good understanding of language constructs, and be capable of authoring a flow definition.

3.2. What is a flow?

A flow encapsulates a reusable sequence of steps that can execute in different contexts. Below is a Garrett Information Architecture diagram illustrating a reference to a flow that encapsulates the steps of a hotel booking process:

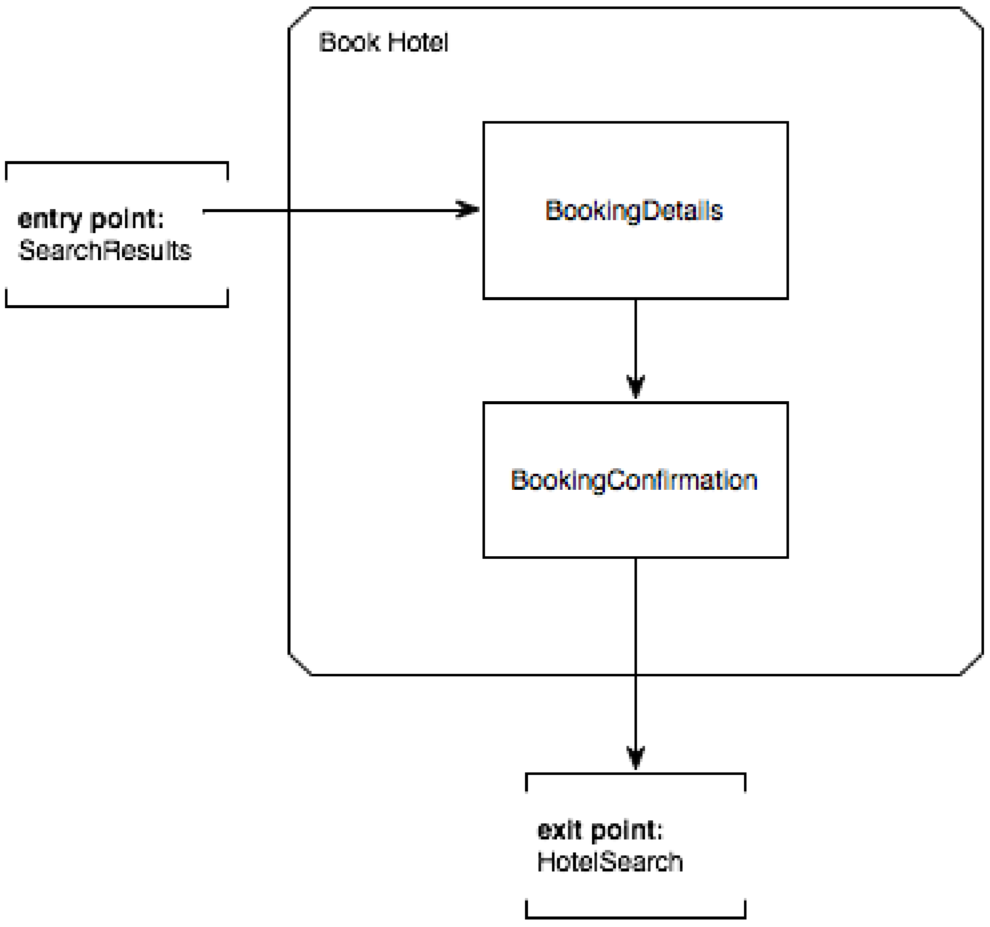


Site Map illustrating a reference to a flow

3.3. What is the makeup of a typical flow?

In Spring Web Flow, a flow consists of a series of steps called "states". Entering a state typically results in a view being displayed to the user. On that view, user events occur that are handled by the state. These events can trigger transitions to other states which result in view navigations.

The example below shows the structure of the book hotel flow referenced in the previous diagram:



Flow diagram

3.4. How are flows authored?

Flows are authored by web application developers using a simple XML-based flow definition language. The next steps of this guide will walk you through the elements of this language.

3.5. Essential language elements

flow

Every flow begins with the following root element:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow.xsd">

</flow>
```

All states of the flow are defined within this element. The first state defined becomes the flow's starting point.

view-state

Use the `view-state` element to define a step of the flow that renders a view:

```
<view-state id="enterBookingDetails" />
```

By convention, a view-state maps its id to a view template in the directory where the flow is located. For example, the state above might render `/WEB-INF/hotels/booking/enterBookingDetails.xhtml` if the flow itself was located in the `/WEB-INF/hotels/booking` directory.

transition

Use the `transition` element to handle events that occur within a state:

```
<view-state id="enterBookingDetails">
    <transition on="submit" to="reviewBooking" />
</view-state>
```

These transitions drive view navigations.

end-state

Use the `end-state` element to define a flow outcome:

```
<end-state id="bookingCancelled" />
```

When a flow transitions to a end-state it terminates and the outcome is returned.

Checkpoint: Essential language elements

With the three elements `view-state`, `transition`, and `end-state`, you can quickly express your view navigation logic. Teams often do this before adding flow behaviors so they can focus on developing the user interface of the application with end users first. Below is a sample flow that implements its view navigation logic using these elements:

```
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow.xsd">

    <view-state id="enterBookingDetails">
        <transition on="submit" to="reviewBooking" />
    </view-state>

    <view-state id="reviewBooking">
        <transition on="confirm" to="bookingConfirmed" />
        <transition on="revise" to="enterBookingDetails" />
        <transition on="cancel" to="bookingCancelled" />
    </view-state>

    <end-state id="bookingConfirmed" />

    <end-state id="bookingCancelled" />

</flow>
```

3.6. Actions

Most flows need to express more than just view navigation logic. Typically they also need to invoke business services of the application or other actions.

Within a flow, there are several points where you can execute actions. These points are:

- On flow start
- On state entry
- On view render
- On transition execution
- On state exit
- On flow end

Actions are defined using a concise expression language. Spring Web Flow uses the Unified EL by default. The next few sections will cover the essential language elements for defining actions.

evaluate

The action element you will use most often is the `evaluate` element. Use the `evaluate` element to evaluate an expression at a point within your flow. With this single tag you can invoke methods on Spring beans or any other flow variable. For example:

```
<evaluate expression="entityManager.persist(booking)" />
```

Assigning an evaluate result

If the expression returns a value, that value can be saved in the flow's data model called `flowScope`:

```
<evaluate expression="bookingService.findHotels(searchCriteria)" result="flowScope.hotels" />
```

Converting an evaluate result

If the expression returns a value that may need to be converted, specify the desired type using the `result-type` attribute:

```
<evaluate expression="bookingService.findHotels(searchCriteria)" result="flowScope.hotels"
          result-type="dataModel"/>
```

Checkpoint: flow actions

Now review the sample booking flow with actions added:

```
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow.xsd">

    <input name="hotelId" />

    <on-start>
        <evaluate expression="bookingService.createBooking(hotelId, currentUser.name)"
                  result="flowScope.booking" />
    </on-start>

    <view-state id="enterBookingDetails">
        <transition on="submit" to="reviewBooking" />
    </view-state>

    <view-state id="reviewBooking">
        <transition on="confirm" to="bookingConfirmed" />
        <transition on="revise" to="enterBookingDetails" />
        <transition on="cancel" to="bookingCancelled" />
    </view-state>

    <end-state id="bookingConfirmed" />

    <end-state id="bookingCancelled" />

</flow>
```

This flow now creates a Booking object in flow scope when it starts. The id of the hotel to book is obtained from a flow input attribute.

3.7. Input/Output Mapping

Each flow has a well-defined input/output contract. Flows can be passed input attributes when they start, and can return output attributes when they end. In this respect, calling a flow is conceptually similar to calling a method with the following signature:

```
FlowOutcome flowId(Map<String, Object> inputAttributes);
```

... where a `FlowOutcome` has the following signature:

```
public interface FlowOutcome {
   public String getName();
   public Map<String, Object> getOutputAttributes();
}
```

input

Use the `input` element to declare a flow input attribute:

```
<input name="hotelId" />
```

Input values are saved in flow scope under the name of the attribute. For example, the input above would be saved under the name `hotelId`.

Declaring an input type

Use the `type` attribute to declare the input attribute's type:

```
<input name="hotelId" type="long" />
```

If an input value does not match the declared type, a type conversion will be attempted.

Assigning an input value

Use the `value` attribute to specify an expression to assign the input value to:

```
<input name="hotelId" value="flowScope.myParameterObject.hotelId" />
```

If the expression's value type can be determined, that metadata will be used for type coersion if no `type` attribute is specified.

Marking an input as required

Use the `required` attribute to enforce the input is not null or empty:

```
<input name="hotelId" type="long" value="flowScope.hotelId" required="true" />
```

output

Use the `output` element to declare a flow output attribute. Output attributes are declared within end-states that represent specific flow outcomes.

```
<end-state id="bookingConfirmed">
    <output name="bookingId" />
</end-state>
```

Output values are obtained from flow scope under the name of the attribute. For example, the output above would be assigned the value of the `bookingId` variable.

Specifying the source of an output value

Use the `value` attribute to denote a specific output value expression:

```
<output name="confirmationNumber" value="booking.confirmationNumber" />
```

Checkpoint: input/output mapping

Now review the sample booking flow with input/output mapping:

```
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow.xsd">

    <input name="hotelId" />

    <on-start>
        <evaluate expression="bookingService.createBooking(hotelId, currentUser.name)"
                  result="flowScope.booking" />
    </on-start>

    <view-state id="enterBookingDetails">
        <transition on="submit" to="reviewBooking" />
    </view-state>

    <view-state id="reviewBooking">
        <transition on="confirm" to="bookingConfirmed" />
        <transition on="revise" to="enterBookingDetails" />
        <transition on="cancel" to="bookingCancelled" />
    </view-state>

    <end-state id="bookingConfirmed" >
        <output name="bookingId" value="booking.id"/>
    </end-state>

    <end-state id="bookingCancelled" />

</flow>
```

The flow now accepts a `hotelId` input attribute and returns a `bookingId` output attribute when a new booking is confirmed.

3.8. Variables

A flow may declare one or more instance variables. These variables are allocated when the flow starts. Any `@Autowired` transient references the variable holds are also rewired when the flow resumes.

var

Use the `var` element to declare a flow variable:

```
<var name="searchCriteria" class="com.mycompany.myapp.hotels.search.SearchCriteria"/>
```

Make sure your variable's class implements `java.io.Serializable`, as the instance state is saved between flow requests.

3.9. Variable Scopes

Web Flow can store variables in one of several scopes:

Flow Scope

Flow scope gets allocated when a flow starts and destroyed when the flow ends. With the default implementation, any objects stored in flow scope need to be Serializable.

View Scope

View scope gets allocated when a `view-state` enters and destroyed when the state exits. View scope is *only* referenceable from within a `view-state`. With the default implementation, any objects stored in view scope need to be Serializable.

Request Scope

Request scope gets allocated when a flow is called and destroyed when the flow returns.

Flash Scope

Flash scope gets allocated when a flow starts, cleared after every view render, and destroyed when the flow ends. With the default implementation, any objects stored in flash scope need to be Serializable.

Conversation Scope

Conversation scope gets allocated when a top-level flow starts and destroyed when the top-level flow ends. Conversation scope is shared by a top-level flow and all of its subflows. With the default implementation, conversation scoped objects are stored in the HTTP session and should generally be Serializable to account for typical session replication.

The scope to use is often determined contextually, for example depending on where a variable is defined -- at the start of the flow definition (flow scope), inside a a view state (view scope), etc. In other cases, for example in EL expressions and Java code, it needs to be specified explicitly. Subsequent sections explain how this is done.

3.10. Calling subflows

A flow may call another flow as a subflow. The flow will wait until the subflow returns, then respond to the subflow outcome.

subflow-state

Use the `subflow-state` element to call another flow as a subflow:

```
<subflow-state id="addGuest" subflow="createGuest">
    <transition on="guestCreated" to="reviewBooking">
        <evaluate expression="booking.guests.add(currentEvent.attributes.guest)" />
    </transition>
    <transition on="creationCancelled" to="reviewBooking" />
</subflow-state>
```

The above example calls the `createGuest` flow, then waits for it to return. When the flow returns with a `guestCreated` outcome, the new guest is added to the booking's guest list.

Passing a subflow input

Use the `input` element to pass input to the subflow:

```
<subflow-state id="addGuest" subflow="createGuest">
    <input name="booking" />
    <transition to="reviewBooking" />
</subflow-state>
```

Mapping subflow output

When a subflow completes, its end-state id is returned to the calling flow as the event to use to continue navigation.

The subflow can also create output attributes to which the calling flow can refer within an outcome transition as follows:

```
<transition on="guestCreated" to="reviewBooking">
    <evaluate expression="booking.guests.add(currentEvent.attributes.guest)" />
</transition>
```

In the above example, `guest` is the name of an output attribute returned by the `guestCreated` outcome.

Checkpoint: calling subflows

Now review the sample booking flow calling a subflow:

```
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow.xsd">

    <input name="hotelId" />

    <on-start>
        <evaluate expression="bookingService.createBooking(hotelId, currentUser.name)"
                  result="flowScope.booking" />
    </on-start>

    <view-state id="enterBookingDetails">
        <transition on="submit" to="reviewBooking" />
    </view-state>

    <view-state id="reviewBooking">
        <transition on="addGuest" to="addGuest" />
        <transition on="confirm" to="bookingConfirmed" />
        <transition on="revise" to="enterBookingDetails" />
        <transition on="cancel" to="bookingCancelled" />
    </view-state>

    <subflow-state id="addGuest" subflow="createGuest">
        <transition on="guestCreated" to="reviewBooking">
            <evaluate expression="booking.guests.add(currentEvent.attributes.guest)" />
        </transition>
        <transition on="creationCancelled" to="reviewBooking" />
    </subflow-state>

    <end-state id="bookingConfirmed" >
        <output name="bookingId" value="booking.id" />
    </end-state>

    <end-state id="bookingCancelled" />

</flow>
```

The flow now calls a `createGuest` subflow to add a new guest to the guest list.

4. Expression Language (EL)

4.1. Introduction

Web Flow uses EL to access its data model and to invoke actions. This chapter will familiarize you with EL syntax, configuration, and special EL variables you can reference from your flow definition.

EL is used for many things within a flow including:

1. Access client data such as declaring flow inputs or referencing request parameters.

2. Access data in Web Flow's `RequestContext` such as `flowScope` or `currentEvent`.

3. Invoke methods on Spring-managed objects through actions.

4. Resolve expressions such as state transition criteria, subflow ids, and view names.

EL is also used to bind form parameters to model objects and reversely to render formatted form fields from the properties of a model object. That however does not apply when using Web Flow with JSF in which case the standard JSF component lifecycle applies.

Expression types

An important concept to understand is there are two types of expressions in Web Flow: standard expressions and template expressions.

Standard Expressions

The first and most common type of expression is the *standard expression*. Such expressions are evaluated directly by the EL and need not be enclosed in delimiters like `#{}`. For example:

```
<evaluate expression="searchCriteria.nextPage()" />
```

The expression above is a standard expression that invokes the `nextPage` method on the `searchCriteria` variable when evaluated. If you attempt to enclose this expression in a special delimiter like `#{}` you will get an `IllegalArgumentException`. In this context the delimiter is seen as redundant. The only acceptable value for the `expression` attribute is an single expression string.

Template expressions

The second type of expression is a *template expression*. A template expression allows mixing of literal text with one or more standard expressions. Each standard expression block is explicitly surrounded with the `#{}` delimiters. For example:

```
<view-state id="error" view="error-#{externalContext.locale}.xhtml" />
```

The expression above is a template expression. The result of evaluation will be a string that concatenates literal text such as `error-` and `.xhtml` with the result of evaluating `externalContext.locale`. As you can see, explicit delimiters are necessary here to demarcate standard expression blocks within the template.

Note

See the Web Flow XML schema for a complete listing of those XML attributes that accept standard expressions and those that accept template expressions. You can also use F2 in Eclipse (or equivalent shortcut in other IDEs) to access available documentation when typing out specific flow definition attributes.

4.2. EL Implementations

Spring EL

Web Flow uses the Spring Expression Language (Spring EL). Spring EL was created to provide a single, well-supported expression language for use across all the products in the Spring portfolio. It is distributed as a separate jar `org.springframework.expression` in the Spring Framework.

Unified EL

Use of Unified EL also implies a dependency on `el-api` although that is typically *provided* by your web container. Although Spring EL is the default and recommended expression language to use, it is possible to replace it with Unified EL if you wish to do so. You need the following Spring configuration to plug in the `WebFlowELExpressionParser` to the `flow-builder-services`:

```
<webflow:flow-builder-services expression-parser="expressionParser"/>

<bean id="expressionParser" class="org.springframework.webflow.expression.el.WebFlowELExpressionParser">
    <constructor-arg>
        <bean class="org.jboss.el.ExpressionFactoryImpl" />
    </constructor-arg>
</bean>
```

Note that if your application is registering custom converters it's important to ensure the WebFlowELExpressionParser is configured with the conversion service that has those custom converters.

```
<webflow:flow-builder-services expression-parser="expressionParser" conversion-
service="conversionService"/>

<bean id="expressionParser" class="org.springframework.webflow.expression.el.WebFlowELExpressionParser">
    <constructor-arg>
        <bean class="org.jboss.el.ExpressionFactoryImpl" />
    </constructor-arg>
    <property name="conversionService" ref="conversionService"/>
</bean>

<bean id="conversionService" class="somepackage.ApplicationConversionService"/>
```

4.3. EL portability

In general, you will find Spring EL and Unified EL to have a very similar syntax.

Note however there are some advantages to Spring EL. For example Spring EL is closely integrated with the type conversion of Spring 3 and that allows you to take full advantage of its features. Specifically the automatic detection of generic types as well as the use of formatting annotations is currently supported with Spring EL only.

There are some minor changes to keep in mind when upgrading to Spring EL from Unified EL as follows:

1. Expressions deliniated with `${}` in flow definitions must be changed to `#{}`.
2. Expressions testing the current event `#{currentEvent == 'submit'}` must be changed to `#{currentEvent.id == 'submit'}`.
3. Resolving properties such as `#{currentUser.name}` may cause NullPointerException without any checks such as `#{currentUser != null ? currentUser.name : null}`. A much better alternative though is the safe navigation operator `#{currentUser?.name}`.

For more information on Spring EL syntax please refer to the Language Reference section in the Spring Documentation.

4.4. Special EL variables

There are several implicit variables you may reference from within a flow. These variables are discussed in this section.

Keep in mind this general rule. Variables referring to data scopes (flowScope, viewScope, requestScope, etc.) should only be used when assigning a new variable to one of the scopes.

For example when assigning the result of the call to `bookingService.findHotels(searchCriteria)` to a new variable called "hotels" you must prefix it with a scope variable in order to let Web Flow know where you want it stored:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow" ... >

 <var name="searchCriteria" class="org.springframework.webflow.samples.booking.SearchCriteria" />

 <view-state id="reviewHotels">
  <on-render>
   <evaluate expression="bookingService.findHotels(searchCriteria)" result="viewScope.hotels" />
  </on-render>
 </view-state>

</flow>
```

However when setting an existing variable such as "searchCriteria" in the example below, you reference the variable directly without prefixing it with any scope variables:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow" ... >

 <var name="searchCriteria" class="org.springframework.webflow.samples.booking.SearchCriteria" />

 <view-state id="reviewHotels">
  <transition on="sort">
   <set name="searchCriteria.sortBy" value="requestParameters.sortBy" />
  </transition>
 </view-state>

</flow>
```

The following is the list of implicit variables you can reference within a flow definition:

flowScope

Use `flowScope` to assign a flow variable. Flow scope gets allocated when a flow starts and destroyed when the flow ends. With the default implementation, any objects stored in flow scope need to be Serializable.

```
<evaluate expression="searchService.findHotel(hotelId)" result="flowScope.hotel" />
```

viewScope

Use `viewScope` to assign a view variable. View scope gets allocated when a `view-state` enters and destroyed when the state exits. View scope is *only* referenceable from within a `view-state`. With the default implementation, any objects stored in view scope need to be Serializable.

```
<on-render>
    <evaluate expression="searchService.findHotels(searchCriteria)" result="viewScope.hotels"
              result-type="dataModel" />
</on-render>
```

requestScope

Use `requestScope` to assign a request variable. Request scope gets allocated when a flow is called and destroyed when the flow returns.

```
<set name="requestScope.hotelId" value="requestParameters.id" type="long" />
```

flashScope

Use `flashScope` to assign a flash variable. Flash scope gets allocated when a flow starts, cleared after every view render, and destroyed when the flow ends. With the default implementation, any objects stored in flash scope need to be Serializable.

```
<set name="flashScope.statusMessage" value="'Booking confirmed'" />
```

conversationScope

Use `conversationScope` to assign a conversation variable. Conversation scope gets allocated when a top-level flow starts and destroyed when the top-level flow ends. Conversation scope is shared by a top-level flow and all of its subflows. With the default implementation, conversation scoped objects are stored in the HTTP session and should generally be Serializable to account for typical session replication.

```
<evaluate expression="searchService.findHotel(hotelId)" result="conversationScope.hotel" />
```

requestParameters

Use `requestParameters` to access a client request parameter:

```
<set name="requestScope.hotelId" value="requestParameters.id" type="long" />
```

currentEvent

Use `currentEvent` to access attributes of the current `Event`:

```
<evaluate expression="booking.guests.add(currentEvent.attributes.guest)" />
```

currentUser

Use `currentUser` to access the authenticated `Principal`:

```
<evaluate expression="bookingService.createBooking(hotelId, currentUser.name)"
          result="flowScope.booking" />
```

messageContext

Use `messageContext` to access a context for retrieving and creating flow execution messages, including error and success messages. See the `MessageContext` Javadocs for more information.

```
<evaluate expression="bookingValidator.validate(booking, messageContext)" />
```

resourceBundle

Use `resourceBundle` to access a message resource.

```
<set name="flashScope.successMessage" value="resourceBundle.successMessage" />
```

flowRequestContext

Use `flowRequestContext` to access the `RequestContext` API, which is a representation of the current flow request. See the API Javadocs for more information.

flowExecutionContext

Use `flowExecutionContext` to access the `FlowExecutionContext` API, which is a representation of the current flow state. See the API Javadocs for more information.

flowExecutionUrl

Use `flowExecutionUrl` to access the context-relative URI for the current flow execution view-state.

externalContext

Use `externalContext` to access the client environment, including user session attributes. See the `ExternalContext` API JavaDocs for more information.

```
<evaluate expression="searchService.suggestHotels(externalContext.sessionMap.userProfile)"
          result="viewScope.hotels" />
```

4.5. Scope searching algorithm

As mentioned earlier in this section when assigning a variable in one of the flow scopes, referencing that scope is required. For example:

```
<set name="requestScope.hotelId" value="requestParameters.id" type="long" />
```

When simply accessing a variable in one of the scopes, referencing the scope is optional. For example:

```
<evaluate expression="entityManager.persist(booking)" />
```

When no scope is specified, like in the use of `booking` above, a scope searching algorithm is used. The algorithm will look in request, flash, view, flow, and conversation scope for the variable. If no such variable is found, an `EvaluationException` will be thrown.

5. Rendering views

5.1. Introduction

This chapter shows you how to use the `view-state` element to render views within a flow.

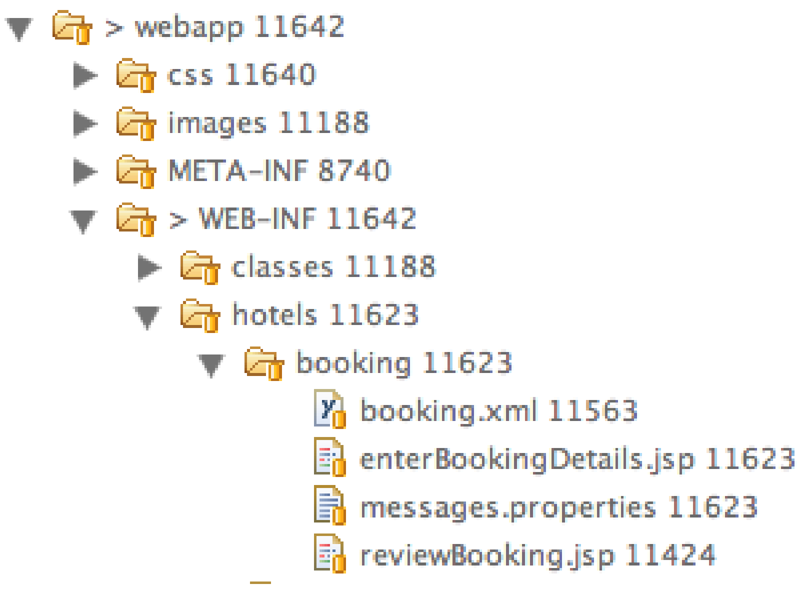
5.2. Defining view states

Use the `view-state` element to define a step of the flow that renders a view and waits for a user event to resume:

```
<view-state id="enterBookingDetails">
    <transition on="submit" to="reviewBooking" />
</view-state>
```

By convention, a view-state maps its id to a view template in the directory where the flow is located. For example, the state above might render `/WEB-INF/hotels/booking/enterBookingDetails.xhtml` if the flow itself was located in the `/WEB-INF/hotels/booking` directory.

Below is a sample directory structure showing views and other resources like message bundles co-located with their flow definition:



Flow Packaging

5.3. Specifying view identifiers

Use the `view` attribute to specify the id of the view to render explicitly.

Flow relative view ids

The view id may be a relative path to view resource in the flow's working directory:

```
<view-state id="enterBookingDetails" view="bookingDetails.xhtml">
```

Absolute view ids

The view id may be a absolute path to a view resource in the webapp root directory:

```
<view-state id="enterBookingDetails" view="/WEB-INF/hotels/booking/bookingDetails.xhtml">
```

Logical view ids

With some view frameworks, such as Spring MVC's view framework, the view id may also be a logical identifier resolved by the framework:

```
<view-state id="enterBookingDetails" view="bookingDetails">
```

See the Spring MVC integration section for more information on how to integrate with the MVC `ViewResolver` infrastructure.

5.4. View scope

A view-state allocates a new `viewScope` when it enters. This scope may be referenced within the view-state to assign variables that should live for the duration of the state. This scope is useful for manipulating objects over a series of requests from the same view, often Ajax requests. A view-state destroys its viewScope when it exits.

Allocating view variables

Use the `var` tag to declare a view variable. Like a flow variable, any `@Autowired` references are automatically restored when the view state resumes.

```
<var name="searchCriteria" class="com.mycompany.myapp.hotels.SearchCriteria" />
```

Assigning a viewScope variable

Use the `on-render` tag to assign a variable from an action result before the view renders:

```
<on-render>
    <evaluate expression="bookingService.findHotels(searchCriteria)" result="viewScope.hotels" />
</on-render>
```

Manipulating objects in view scope

Objects in view scope are often manipulated over a series of requests from the same view. The following example pages through a search results list. The list is updated in view scope before each render. Asynchronous event handlers modify the current data page, then request re-rendering of the search results fragment.

```
<view-state id="searchResults">
    <on-render>
        <evaluate expression="bookingService.findHotels(searchCriteria)"
                  result="viewScope.hotels" />
    </on-render>
    <transition on="next">
        <evaluate expression="searchCriteria.nextPage()" />
        <render fragments="searchResultsFragment" />
    </transition>
    <transition on="previous">
        <evaluate expression="searchCriteria.previousPage()" />
        <render fragments="searchResultsFragment" />
    </transition>
</view-state>
```

5.5. Executing render actions

Use the `on-render` element to execute one or more actions before view rendering. Render actions are executed on the initial render as well as any subsequent refreshes, including any partial re-renderings of the view.

```
<on-render>
    <evaluate expression="bookingService.findHotels(searchCriteria)" result="viewScope.hotels" />
</on-render>
```

5.6. Binding to a model

Use the `model` attribute to declare a model object the view binds to. This attribute is typically used in conjunction with views that render data controls, such as forms. It enables form data binding and validation behaviors to be driven from metadata on your model object.

The following example declares an `enterBookingDetails` state manipulates the `booking` model:

```
<view-state id="enterBookingDetails" model="booking">
```

The model may be an object in any accessible scope, such as `flowScope` or `viewScope`. Specifying a `model` triggers the following behavior when a view event occurs:

1. View-to-model binding. On view postback, user input values are bound to model object properties for you.

2. Model validation. After binding, if the model object requires validation that validation logic will be invoked.

For a flow event to be generated that can drive a view state transition, model binding must complete successfully. If model binding fails, the view is re-rendered to allow the user to revise their edits.

5.7. Performing type conversion

When request parameters are used to populate the model (commonly referred to as data binding), type conversion is required to parse String-based request parameter values before setting target model properties. Default type conversion is available for many common Java types such as numbers, primitives, enums, and Dates. Users also have the ability to register their own type conversion logic for user-defined types, and to override the default Converters.

Type Conversion Options

Starting with version 2.1 Spring Web Flow uses the type conversion and formatting system introduced in Spring 3 for nearly all type conversion needs. Previously Web Flow applications used a type conversion mechanism that was different from the one in Spring MVC, which relied on the `java.beans.PropertyEditor` abstraction. Spring 3 offers a modern type conversion alternative to PropertyEditors that was actually influenced by Web Flow's own type conversion system. Hence Web Flow users should find it natural to work with the new Spring 3 type conversion. Another obvious and very important benefit of this change is that a single type conversion mechanism can now be used across Spring MVC And Spring Web Flow.

Upgrading to Spring 3 Type Conversion And Formatting

What does this practically mean for existing applications? Existing applications are likely registering their own converters of type `org.springframework.binding.convert.converters.Converter` through a sub-class of `DefaultConversionService` available in Spring Binding. Those converters can continue to be registered as before. They will be adapted as Spring 3 `GenericConverter` types and registered with a Spring 3 `org.springframework.core.convert.ConversionService` instance. In other words existing converters will be invoked through Spring's type conversion service.

The only exception to this rule are named converters, which can be referenced from a `binding` element in a `view-state`:

```
public class ApplicationConversionService extends DefaultConversionService {
    public ApplicationConversionService() {
        addDefaultConverters();
        addDefaultAliases();
        addConverter("customConverter", new CustomConverter());
    }
}
```

```
<view-state id="enterBookingDetails" model="booking">
    <binder>
        <binding property="checkinDate" required="true" converter="customConverter" />
    </binder>
</view-state>
```

Named converters are not supported and cannot be used with the type conversion service available in Spring 3. Therefore such converters will not be adapted and will continue to work as before, i.e. will not involve the Spring 3 type conversion. However, this mechanism is deprecated and applications are encouraged to favor Spring 3 type conversion and formatting features.

Also note that the existing Spring Binding `DefaultConversionService` no longer registers any default converters. Instead Web Flow now relies on the default type converters and formatters in Spring 3.

In summary the Spring 3 type conversion and formatting is now used almost exclusively in Web Flow. Although existing applications will work without any changes, we encourage moving towards unifying the type conversion needs of Spring MVC and Spring Web Flow parts of applications.

Configuring Type Conversion and Formatting

In Spring MVC an instance of a `FormattingConversionService` is created automatically through the custom MVC namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

 <mvc:annotation-driven/>
```

Internally that is done with the help of `FormattingConversionServiceFactoryBean`, which registers a default set of converters and formatters. You can customize the conversion service instance used in Spring MVC through the `conversion-service` attribute:

```
<mvc:annotation-driven conversion-service="applicationConversionService" />
```

In Web Flow an instance of a Spring Binding `DefaultConversionService` is created automatically, which does not register any converters. Instead it delegates to a `FormattingConversionService` instance for all type conversion needs. By default this is not the same `FormattingConversionService` instance as the one used in Spring 3. However that won't make a practical difference until you start registering your own formatters.

The `DefaultConversionService` used in Web Flow can be customized through the flow-builder-services element:

```
<webflow:flow-builder-services id="flowBuilderServices" conversion-service="defaultConversionService" />
```

Connecting the dots in order to register your own formatters for use in both Spring MVC and in Spring Web Flow you can do the following. Create a class to register your custom formatters:

```
public class ApplicationConversionServiceFactoryBean extends FormattingConversionServiceFactoryBean {

    @Override
    protected void installFormatters(FormatterRegistry registry) {
        // ...
    }

}
```

Configure it for use in Spring MVC:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <mvc:annotation-driven conversion-service="applicationConversionService" />

    <!--
     Alternatively if you prefer annotations for DI:
       1. Add @Component to the factory bean.
       2. Add a component-scan element (from the context custom namespace) here.
       3. Remove XML bean declaration below.
      -->

    <bean id="applicationConversionService" class="somepackage.ApplicationConversionServiceFactoryBean">
```

Connection the Web Flow `DefaultConversionService` to the same "applicationConversionService" bean used in Spring MVC:

```
    <webflow:flow-registry id="flowRegistry" flow-builder-services="flowBuilderServices" ... />

    <webflow:flow-builder-services id="flowBuilderServices" conversion-
service="defaultConversionService" ... />


  <bean id="defaultConversionService" class="org.springframework.binding.convert.service.DefaultConversionService">
     <constructor-arg ref="applicationConversionSevice"/>
    </bean>
```

Of course it is also possible to mix and match. Register new Spring 3 `Formatter` types through the "applicationConversionService". Register existing Spring Binding `Converter` types through the "defaultConversionService".

Working With Spring 3 Type Conversion And Formatting

An important concept to understand is the difference between type converters and formatters.

Type converters in Spring 3, provided in `org.springframework.core`, are for general-purpose type conversion between any two object types. In addition to the most simple `Converter` type, two other interfaces are `ConverterFactory` and `GenericConverter`.

Formatters in Spring 3, provided in `org.springframework.context`, have the more specialized purpose of representing Objects as Strings. The `Formatter` interface extends the `Printer` and `Parser` interfaces for converting an Object to a String and turning a String into an Object.

Web developers will find the `Formatter` interface most relevant because it fits the needs of web applications for type conversion.

> **Note**
>
> An important point to be made is that Object-to-Object conversion is a generalization of the more specific Object-to-String conversion. In fact in the end `Formatters` are reigstered as

> `GenericConverter` types with Spring's `GenericConversionService` making them equal to any other converter.

Formatting Annotations

One of the best features of the new type conversion is the ability to use annotations for a better control over formatting in a concise manner. Annotations can be placed on model attributes and on arguments of @Controller methods that are mapped to requests. Out of the box Spring provides two annotations `NumberFormat` and `DateTimeFormat` but you can create your own and have them registered along with the associated formatting logic. You can see examples of the `DateTimeFormat` annotation in the Spring Travel and in the Petcare along with other samples in the Spring Samples repository.

Working With Dates

The `DateTimeFormat` annotation implies use of Joda Time. If that is present on the classpath the use of this annotation is enabled automatically. By default neither Spring MVC nor Web Flow register any other date formatters or converters. Therefore it is important for applications to register a custom formatter to specify the default way for printing and parsing dates. The `DateTimeFormat` annotation on the other hand provides more fine-grained control where it is necessary to deviate from the default.

For more information on working with Spring 3 type conversion and formatting please refer to the relevant sections of the Spring documentation.

5.8. Suppressing binding

Use the `bind` attribute to suppress model binding and validation for particular view events. The following example suppresses binding when the `cancel` event occurs:

```
<view-state id="enterBookingDetails" model="booking">
    <transition on="proceed" to="reviewBooking">
    <transition on="cancel" to="bookingCancelled" bind="false" />
</view-state>
```

5.9. Specifying bindings explicitly

Use the `binder` element to configure the exact set of model properties to apply data binding to. This is useful to restrict the set of "allowed fields" per view. Not using this could lead to a security issue, depending on the application domain and actual users, since by default if the binder element is not specified all public properties of the model are eligible for data binding by the view. By contrast when the `binder` element is specified, only the explicitly configured bindings are allowed. Below is an example:

```
<view-state id="enterBookingDetails" model="booking">
    <binder>
        <binding property="creditCard" />
        <binding property="creditCardName" />
        <binding property="creditCardExpiryMonth" />
        <binding property="creditCardExpiryYear" />
    </binder>
    <transition on="proceed" to="reviewBooking" />
    <transition on="cancel" to="cancel" bind="false" />
</view-state>
```

Each binding may also apply a converter to format the model property value for display in a custom manner. If no converter is specified, the default converter for the model property's type will be used.

```
<view-state id="enterBookingDetails" model="booking">
    <binder>
        <binding property="checkinDate" converter="shortDate" />
        <binding property="checkoutDate" converter="shortDate" />
        <binding property="creditCard" />
        <binding property="creditCardName" />
        <binding property="creditCardExpiryMonth" />
        <binding property="creditCardExpiryYear" />
    </binder>
    <transition on="proceed" to="reviewBooking" />
    <transition on="cancel" to="cancel" bind="false" />
</view-state>
```

In the example above, the `shortDate` converter is bound to the `checkinDate` and `checkoutDate` properties. Custom converters may be registered with the application's ConversionService.

Each binding may also apply a required check that will generate a validation error if the user provided value is null on form postback:

```
<view-state id="enterBookingDetails" model="booking">
    <binder>
        <binding property="checkinDate" converter="shortDate" required="true" />
        <binding property="checkoutDate" converter="shortDate" required="true" />
        <binding property="creditCard" required="true" />
        <binding property="creditCardName" required="true" />
        <binding property="creditCardExpiryMonth" required="true" />
        <binding property="creditCardExpiryYear" required="true" />
    </binder>
    <transition on="proceed" to="reviewBooking">
    <transition on="cancel" to="bookingCancelled" bind="false" />
</view-state>
```

In the example above, all of the bindings are required. If one or more blank input values are bound, validation errors will be generated and the view will re-render with those errors.

5.10. Validating a model

Model validation is driven by constraints specified against a model object. Web Flow supports enforcing such constraints programatically as well as declaratively with JSR-303 Bean Validation annotations.

JSR-303 Bean Validation

Web Flow provides built-in support for the JSR-303 Bean Validation API building on equivalent support available in Spring MVC. To enable JSR-303 validation configure the flow-builder-services with Spring MVC's `LocalValidatorFactoryBean`:

```
<webflow:flow-registry flow-builder-services="flowBuilderServices" />

<webflow:flow-builder-services id="flowBuilderServices" validator="validator" />

<bean id="validator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean" />
```

With the above in place, the configured validator will be applied to all model attributes after data binding.

Note that JSR-303 bean validation and validation by convention (explained in the next section) are not mutually exclusive. In other words Web Flow will apply all available validation mechanisms.

Partial Validation

JSR-303 Bean Validation supports partial validation through validation groups. For example:

```
@NotNull
@Size(min = 2, max = 30, groups = State1.class)
private String name;
```

In a flow definition you can specify validation hints on a view state or on a transition and those will be resolved to validation groups. For example:

```
<view-state id="state1" model="myModel" validation-hints="'group1,group2'">
```

The *validation-hints* attribute is an expression that in the above example resolves to a comma-delimited String consisting of the hints "group1" and "group2". A `ValidationHintResolver` is used to resolve these hints. The `BeanValidationHintResolver` used by default tries to resolve these strings to Class-based bean validation groups. To do that it looks for matching inner types in the model or its parent.

For example given `org.example.MyModel` with inner types `Group1` and `Group2` it is sufficient to supply the simple type names, i.e. "group1" and "group2". You can also provide fully qualified type names.

A hint with the value "default" has a special meaning and is translated to the default validation group in Bean Validation `javax.validation.groups.Default`.

A custom `ValidationHintResolver` can be configured if necessary through the validationHintResolver property of the flow-builder-services element:

```
<webflow:flow-registry flow-builder-services="flowBuilderServices" />

<webflow:flow-builder-services id="flowBuilderServices" validator=".." validation-hint-resolver=".." />
```

Programmatic validation

There are two ways to perform model validation programatically. The first is to implement validation logic in your model object. The second is to implement an external `Validator`. Both ways provide you with a `ValidationContext` to record error messages and access information about the current user.

Implementing a model validate method

Defining validation logic in your model object is the simplest way to validate its state. Once such logic is structured according to Web Flow conventions, Web Flow will automatically invoke that logic during the view-state postback lifecycle. Web Flow conventions have you structure model validation logic by view-state, allowing you to easily validate the subset of model properties that are editable on that view. To do this, simply create a public method with the name `validate${state}`, where `${state}` is the id of your view-state where you want validation to run. For example:

```
public class Booking {
    private Date checkinDate;
    private Date checkoutDate;
    ...

    public void validateEnterBookingDetails(ValidationContext context) {
        MessageContext messages = context.getMessageContext();
        if (checkinDate.before(today())) {
            messages.addMessage(new MessageBuilder().error().source("checkinDate").
                defaultText("Check in date must be a future date").build());
        } else if (!checkinDate.before(checkoutDate)) {
            messages.addMessage(new MessageBuilder().error().source("checkoutDate").
                defaultText("Check out date must be later than check in date").build());
        }
    }
}
```

In the example above, when a transition is triggered in a `enterBookingDetails` view-state that is editing a `Booking` model, Web Flow will invoke the `validateEnterBookingDetails(ValidationContext)` method automatically unless validation has been suppressed for that transition. An example of such a view-state is shown below:

```
<view-state id="enterBookingDetails" model="booking">
    <transition on="proceed" to="reviewBooking">
</view-state>
```

Any number of validation methods are defined. Generally, a flow edits a model over a series of views. In that case, a validate method would be defined for each view-state where validation needs to run.

Implementing a Validator

The second way is to define a separate object, called a *Validator*, which validates your model object. To do this, first create a class whose name has the pattern ${model}Validator, where `${model}` is the capitialized form of the model expression, such as `booking`. Then define a public method with the name `validate${state}`, where `${state}` is the id of your view-state, such as `enterBookingDetails`. The class should then be deployed as a Spring bean. Any number of validation methods can be defined. For example:

```
@Component
public class BookingValidator {
    public void validateEnterBookingDetails(Booking booking, ValidationContext context) {
        MessageContext messages = context.getMessageContext();
        if (booking.getCheckinDate().before(today())) {
            messages.addMessage(new MessageBuilder().error().source("checkinDate").
                defaultText("Check in date must be a future date").build());
        } else if (!booking.getCheckinDate().before(booking.getCheckoutDate())) {
            messages.addMessage(new MessageBuilder().error().source("checkoutDate").
                defaultText("Check out date must be later than check in date").build());
        }
    }
}
```

In the example above, when a transition is triggered in a `enterBookingDetails` view-state that is editing a `Booking` model, Web Flow will invoke the `validateEnterBookingDetails(Booking, ValidationContext)` method automatically unless validation has been suppressed for that transition.

A Validator can also accept a Spring MVC `Errors` object, which is required for invoking existing Spring Validators.

Validators must be registered as Spring beans employing the naming convention `${model}Validator` to be detected and invoked automatically. In the example above, Spring 2.5 classpath-scanning would detect the `@Component` and automatically register it as a bean with the name `bookingValidator`. Then, anytime the `booking` model needs to be validated, this `bookingValidator` instance would be invoked for you.

Default validate method

A *Validator* class can also define a method called `validate` not associated (by convention) with any specific view-state.

```
@Component
public class BookingValidator {
    public void validate(Booking booking, ValidationContext context) {
        //...
    }
}
```

In the above code sample the method `validate` will be called every time a Model of type `Booking` is validated (unless validation has been suppressed for that transition). If needed the default method can also be called in addition to an existing state-specific method. Consider the following example:

```
@Component
public class BookingValidator {
    public void validate(Booking booking, ValidationContext context) {
        //...
    }
    public void validateEnterBookingDetails(Booking booking, ValidationContext context) {
        //...
    }
}
```

In above code sample the method `validateEnterBookingDetails` will be called first. The default `validate` method will be called next.

ValidationContext

A ValidationContext allows you to obtain a `MessageContext` to record messages during validation. It also exposes information about the current user, such as the signaled `userEvent` and the current user's `Principal` identity. This information can be used to customize validation logic based on what button or link was activated in the UI, or who is authenticated. See the API Javadocs for `ValidationContext` for more information.

5.11. Suppressing validation

Use the `validate` attribute to suppress model validation for particular view events:

```
<view-state id="chooseAmenities" model="booking">
    <transition on="proceed" to="reviewBooking">
    <transition on="back" to="enterBookingDetails" validate="false" />
</view-state>
```

In this example, data binding will still occur on `back` but validation will be suppressed.

5.12. Executing view transitions

Define one or more `transition` elements to handle user events that may occur on the view. A transition may take the user to another view, or it may simply execute an action and re-render the current view. A transition may also request the rendering of parts of a view called "fragments" when handling an Ajax event. Finally, "global" transitions that are shared across all views may also be defined.

Implementing view transitions is illustrated in the following sections.

Transition actions

A view-state transition can execute one or more actions before executing. These actions may return an error result to prevent the transition from exiting the current view-state. If an error result occurs, the view will re-render and should display an appropriate message to the user.

If the transition action invokes a plain Java method, the invoked method may return a boolean whose value, true or false, indicates whether the transition should take place or be prevented from executing. A method may also return a String where the literal values "success", "yes", or "true" indicate the transition should occur, and any other value means the opposite. This technique can be used to handle exceptions thrown by service-layer methods. The example below invokes an action that calls a service and handles an exceptional situation:

```
<transition on="submit" to="bookingConfirmed">
    <evaluate expression="bookingAction.makeBooking(booking, messageContext)" />
</transition>
```

```
public class BookingAction {
   public boolean makeBooking(Booking booking, MessageContext context) {
       try {
           bookingService.make(booking);
           return true;
       } catch (RoomNotAvailableException e) {
           context.addMessage(new MessageBuilder().error().
               .defaultText("No room is available at this hotel").build());
           return false;
       }
   }
}
```

When there is more than one action defined on a transition, if one returns an error result the remaining actions in the set will *not* be executed. If you need to ensure one transition action's result cannot impact the execution of another, define a single transition action that invokes a method that encapsulates all the action logic.

Global transitions

Use the flow's `global-transitions` element to create transitions that apply across all views. Global-transitions are often used to handle global menu links that are part of the layout.

```
<global-transitions>
    <transition on="login" to="login" />
    <transition on="logout" to="logout" />
</global-transitions>
```

Event handlers

From a view-state, transitions without targets can also be defined. Such transitions are called "event handlers":

```
<transition on="event">
    <!-- Handle event -->
</transition>
```

These event handlers do not change the state of the flow. They simply execute their actions and re-render the current view or one or more fragments of the current view.

Rendering fragments

Use the `render` element within a transition to request partial re-rendering of the current view after handling the event:

```
<transition on="next">
    <evaluate expression="searchCriteria.nextPage()" />
    <render fragments="searchResultsFragment" />
</transition>
```

The fragments attribute should reference the id(s) of the view element(s) you wish to re-render. Specify multiple elements to re-render by separating them with a comma delimiter.

Such partial rendering is often used with events signaled by Ajax to update a specific zone of the view.

5.13. Working with messages

Spring Web Flow's `MessageContext` is an API for recording messages during the course of flow executions. Plain text messages can be added to the context, as well as internationalized messages resolved by a Spring `MessageSource`. Messages are renderable by views and automatically survive flow execution redirects. Three distinct message severities are provided: `info`, `warning`, and `error`. In addition, a convenient `MessageBuilder` exists for fluently constructing messages.

Adding plain text messages

```
MessageContext context = ...
MessageBuilder builder = new MessageBuilder();
context.addMessage(builder.error().source("checkinDate")
    .defaultText("Check in date must be a future date").build());
context.addMessage(builder.warn().source("smoking")
    .defaultText("Smoking is bad for your health").build());
context.addMessage(builder.info()
    .defaultText("We have processed your reservation - thank you and enjoy your stay").build());
```

Adding internationalized messages

```
MessageContext context = ...
MessageBuilder builder = new MessageBuilder();
context.addMessage(builder.error().source("checkinDate").code("checkinDate.notFuture").build());
context.addMessage(builder.warn().source("smoking").code("notHealthy")
    .resolvableArg("smoking").build());
context.addMessage(builder.info().code("reservationConfirmation").build());
```

Using message bundles

Internationalized messages are defined in message bundles accessed by a Spring `MessageSource`. To create a flow-specific message bundle, simply define `messages.properties` file(s) in your flow's directory. Create a default `messages.properties` file and a .properties file for each additional `Locale` you need to support.

```
#messages.properties
checkinDate=Check in date must be a future date
notHealthy={0} is bad for your health
reservationConfirmation=We have processed your reservation - thank you and enjoy your stay
```

From within a view or a flow, you may also access message resources using the `resourceBundle` EL variable:

```
<h:outputText value="#{resourceBundle.reservationConfirmation}" />
```

Understanding system generated messages

There are several places where Web Flow itself will generate messages to display to the user. One important place this occurs is during view-to-model data binding. When a binding error occurs, such as a type conversion error, Web Flow will map that error to a message retrieved from your resource bundle automatically. To lookup the message to display, Web Flow tries resource keys that contain the binding error code and target property name.

As an example, consider a binding to a `checkinDate` property of a `Booking` object. Suppose the user typed in a alphabetic string. In this case, a type conversion error will be raised. Web Flow will map the 'typeMismatch' error code to a message by first querying your resource bundle for a message with the following key:

```
booking.checkinDate.typeMismatch
```

The first part of the key is the model class's short name. The second part of the key is the property name. The third part is the error code. This allows for the lookup of a unique message to display to the user when a binding fails on a model property. Such a message might say:

```
booking.checkinDate.typeMismatch=The check in date must be in the format yyyy-mm-dd.
```

If no such resource key can be found of that form, a more generic key will be tried. This key is simply the error code. The field name of the property is provided as a message argument.

```
typeMismatch=The {0} field is of the wrong type.
```

5.14. Displaying popups

Use the `popup` attribute to render a view in a modal popup dialog:

```
<view-state id="changeSearchCriteria" view="enterSearchCriteria.xhtml" popup="true">
```

When using Web Flow with the Spring Javascript, no client side code is necessary for the popup to display. Web Flow will send a response to the client requesting a redirect to the view from a popup, and the client will honor the request.

5.15. View backtracking

By default, when you exit a view state and transition to a new view state, you can go back to the previous state using the browser back button. These view state history policies are configurable on a per-transition basis by using the `history` attribute.

Discarding history

Set the history attribute to `discard` to prevent backtracking to a view:

```
<transition on="cancel" to="bookingCancelled" history="discard">
```

Invalidating history

Set the history attribute to `invalidate` to prevent backtracking to a view as well all previously displayed views:

```
<transition on="confirm" to="bookingConfirmed" history="invalidate">
```

6. Executing actions

6.1. Introduction

This chapter shows you how to use the `action-state` element to control the execution of an action at a point within a flow. It will also show how to use the `decision-state` element to make a flow routing decision. Finally, several examples of invoking actions from the various points possible within a flow will be discussed.

6.2. Defining action states

Use the `action-state` element when you wish to invoke an action, then transition to another state based on the action's outcome:

```
<action-state id="moreAnswersNeeded">
 <evaluate expression="interview.moreAnswersNeeded()" />
 <transition on="yes" to="answerQuestions" />
 <transition on="no" to="finish" />
</action-state>
```

The full example below illustrates a interview flow that uses the action-state above to determine if more answers are needed to complete the interview:

```
<flow xmlns="http://www.springframework.org/schema/webflow"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/webflow
        http://www.springframework.org/schema/webflow/spring-webflow.xsd">

 <on-start>
  <evaluate expression="interviewFactory.createInterview()" result="flowScope.interview" />
 </on-start>

 <view-state id="answerQuestions" model="questionSet">
  <on-entry>
   <evaluate expression="interview.getNextQuestionSet()" result="viewScope.questionSet" />
  </on-entry>
  <transition on="submitAnswers" to="moreAnswersNeeded">
   <evaluate expression="interview.recordAnswers(questionSet)" />
  </transition>
 </view-state>

 <action-state id="moreAnswersNeeded">
  <evaluate expression="interview.moreAnswersNeeded()" />
  <transition on="yes" to="answerQuestions" />
  <transition on="no" to="finish" />
 </action-state>

 <end-state id="finish" />

</flow>
```

After the execution of each action, the action-state checks the result to see if matches a declared transition to another state. That means if more than one action is configured they are executed in an ordered chain until one returns a result event that matches a state transition out of the action-state while the rest are ignored. This is a form of the Chain of Responsibility (CoR) pattern.

The result of an action's execution is typically the criteria for a transition out of this state. Additional information in the current RequestContext may also be tested as part of custom transitional criteria allowing for sophisticated transition expressions that reason on contextual state.

Note also that an action-state just like any other state can have one more on-entry actions that are executed as a list from start to end.

6.3. Defining decision states

Use the `decision-state` element as an alternative to the action-state to make a routing decision using a convenient if/else syntax. The example below shows the `moreAnswersNeeded` state above now implemented as a decision state instead of an action-state:

```
<decision-state id="moreAnswersNeeded">
 <if test="interview.moreAnswersNeeded()" then="answerQuestions" else="finish" />
</decision-state>
```

6.4. Action outcome event mappings

Actions often invoke methods on plain Java objects. When called from action-states and decision-states, these method return values can be used to drive state transitions. Since transitions are triggered by events, a method return value must first be mapped to an Event object. The following table describes how common return value types are mapped to Event objects:

Table 6.1. Action method return value to event id mappings

Method return type	Mapped Event identifier expression
java.lang.String	the String value
java.lang.Boolean	yes (for true), no (for false)
java.lang.Enum	the Enum name
any other type	success

This is illustrated in the example action state below, which invokes a method that returns a boolean value:

```
<action-state id="moreAnswersNeeded">
 <evaluate expression="interview.moreAnswersNeeded()" />
 <transition on="yes" to="answerQuestions" />
 <transition on="no" to="finish" />
</action-state>
```

6.5. Action implementations

While writing action code as POJO logic is the most common, there are several other action implementation options. Sometimes you need to write action code that needs access to the flow context. You can always invoke a POJO and pass it the flowRequestContext as an EL variable. Alternatively, you may implement the `Action` interface or extend from the `MultiAction` base class. These options provide stronger type safety when you have a natural coupling between your action code and Spring Web Flow APIs. Examples of each of these approaches are shown below.

Invoking a POJO action

```
<evaluate expression="pojoAction.method(flowRequestContext)" />
```

```
public class PojoAction {
 public String method(RequestContext context) {
  ...
 }
}
```

Invoking a custom Action implementation

```
<evaluate expression="customAction" />
```

```
public class CustomAction implements Action {
 public Event execute(RequestContext context) {
  ...
 }
}
```

Invoking a MultiAction implementation

```
<evaluate expression="multiAction.actionMethod1" />
```

```
public class CustomMultiAction extends MultiAction {
 public Event actionMethod1(RequestContext context) {
  ...
 }

 public Event actionMethod2(RequestContext context) {
  ...
 }

 ...
}
```

6.6. Action exceptions

Actions often invoke services that encapsulate complex business logic. These services may throw business exceptions that the action code should handle.

Handling a business exception with a POJO action

The following example invokes an action that catches a business exception, adds a error message to the context, and returns a result event identifier. The result is treated as a flow event which the calling flow can then respond to.

```
<evaluate expression="bookingAction.makeBooking(booking, flowRequestContext)" />
```

```
public class BookingAction {
public String makeBooking(Booking booking, RequestContext context) {
    try {
     BookingConfirmation confirmation = bookingService.make(booking);
     context.getFlowScope().put("confirmation", confirmation);
     return "success";
    } catch (RoomNotAvailableException e) {
     context.addMessage(new MessageBuilder().error().
      .defaultText("No room is available at this hotel").build());
     return "error";
    }
}
}
```

Handling a business exception with a MultiAction

The following example is functionally equivlant to the last, but implemented as a MultiAction instead of a POJO action. The MultiAction requires its action methods to be of the signature `Event ${methodName}(RequestContext)`, providing stronger type safety, while a POJO action allows for more freedom.

```
<evaluate expression="bookingAction.makeBooking" />
```

```
public class BookingAction extends MultiAction {
public Event makeBooking(RequestContext context) {
    try {
     Booking booking = (Booking) context.getFlowScope().get("booking");
     BookingConfirmation confirmation = bookingService.make(booking);
     context.getFlowScope().put("confirmation", confirmation);
     return success();
    } catch (RoomNotAvailableException e) {
     context.getMessageContext().addMessage(new MessageBuilder().error().
      .defaultText("No room is available at this hotel").build());
     return error();
    }
}
}
```

Using an exception-handler element

In general it is recommended to catch exceptions in actions and return result events that drive standard transitions, it is also possible to add an `exception-handler` sub-element to any state type with a `bean` attribute referencing a bean of type `FlowExecutionExceptionHandler`. This is an advanced option that if used incorrectly can leave the flow execution in an invalid state. Consider the build-in `TransitionExecutingFlowExecutionExceptionHandler` as example of a correct implementation.

6.7. Other Action execution examples

on-start

The following example shows an action that creates a new Booking object by invoking a method on a service:

```
<flow xmlns="http://www.springframework.org/schema/webflow"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/webflow
        http://www.springframework.org/schema/webflow/spring-webflow.xsd">

 <input name="hotelId" />

 <on-start>
  <evaluate expression="bookingService.createBooking(hotelId, currentUser.name)"
      result="flowScope.booking" />
 </on-start>

</flow>
```

on-entry

The following example shows a state entry action that sets the special `fragments` variable that causes the view-state to render a partial fragment of its view:

```
<view-state id="changeSearchCriteria" view="enterSearchCriteria.xhtml" popup="true">
 <on-entry>
  <render fragments="hotelSearchForm" />
 </on-entry>
</view-state>
```

on-exit

The following example shows a state exit action that releases a lock on a record being edited:

```
<view-state id="editOrder">
 <on-entry>
  <evaluate expression="orderService.selectForUpdate(orderId, currentUser)"
      result="viewScope.order" />
 </on-entry>
 <transition on="save" to="finish">
  <evaluate expression="orderService.update(order, currentUser)" />
 </transition>
 <on-exit>
  <evaluate expression="orderService.releaseLock(order, currentUser)" />
 </on-exit>
</view-state>
```

on-end

The following example shows the equivalent object locking behavior using flow start and end actions:

```
<flow xmlns="http://www.springframework.org/schema/webflow"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/webflow
        http://www.springframework.org/schema/webflow/spring-webflow.xsd">

 <input name="orderId" />

 <on-start>
  <evaluate expression="orderService.selectForUpdate(orderId, currentUser)"
      result="flowScope.order" />
 </on-start>

 <view-state id="editOrder">
  <transition on="save" to="finish">
   <evaluate expression="orderService.update(order, currentUser)" />
  </transition>
 </view-state>

 <on-end>
  <evaluate expression="orderService.releaseLock(order, currentUser)" />
 </on-end>

</flow>
```

on-render

The following example shows a render action that loads a list of hotels to display before the view is rendered:

```
<view-state id="reviewHotels">
 <on-render>
  <evaluate expression="bookingService.findHotels(searchCriteria)"
      result="viewScope.hotels" result-type="dataModel" />
 </on-render>
 <transition on="select" to="reviewHotel">
  <set name="flowScope.hotel" value="hotels.selectedRow" />
 </transition>
</view-state>
```

on-transition

The following example shows a transition action adds a subflow outcome event attribute to a collection:

```
<subflow-state id="addGuest" subflow="createGuest">
 <transition on="guestCreated" to="reviewBooking">
  <evaluate expression="booking.guestList.add(currentEvent.attributes.newGuest)" />
 </transition>
</subfow-state>
```

Named actions

The following example shows how to execute a chain of actions in an action-state. The name of each action becomes a qualifier for the action's result event.

```
<action-state id="doTwoThings">
 <evaluate expression="service.thingOne()">
  <attribute name="name" value="thingOne" />
 </evaluate>
 <evaluate expression="service.thingTwo()">
  <attribute name="name" value="thingTwo" />
 </evaluate>
 <transition on="thingTwo.success" to="showResults" />
</action-state>
```

In this example, the flow will transition to `showResults` when `thingTwo` completes successfully.

Streaming actions

Sometimes an Action needs to stream a custom response back to the client. An example might be a flow that renders a PDF document when handling a print event. This can be achieved by having the action stream the content then record "Response Complete" status on the ExternalContext. The responseComplete flag tells the pausing view-state not to render the response because another object has taken care of it.

```
<view-state id="reviewItinerary">
 <transition on="print">
  <evaluate expression="printBoardingPassAction" />
 </transition>
</view-state>
```

```
public class PrintBoardingPassAction extends AbstractAction {
 public Event doExecute(RequestContext context) {
  // stream PDF content here...
  // - Access HttpServletResponse by calling context.getExternalContext().getNativeResponse();
  // - Mark response complete by calling context.getExternalContext().recordResponseComplete();
  return success();
 }
}
```

In this example, when the print event is raised the flow will call the printBoardingPassAction. The action will render the PDF then mark the response as complete.

Handling File Uploads

Another common task is to use Web Flow to handle multipart file uploads in combination with Spring MVC's `MultipartResolver`. Once the resolver is set up correctly as described here and the submitting HTML form is configured with `enctype="multipart/form-data"`, you can easily handle the file upload in a transition action.

Note

The file upload example below below is not relevant when using Web Flow with JSF. See Section 13.8, “Handling File Uploads with JSF” for details of how to upload files using JSF.

Given a form such as:

```
<form:form modelAttribute="fileUploadHandler" enctype="multipart/form-data">
 Select file: <input type="file" name="file"/>
 <input type="submit" name="_eventId_upload" value="Upload" />
</form:form>
```

and a backing object for handling the upload such as:

```
package org.springframework.webflow.samples.booking;

import org.springframework.web.multipart.MultipartFile;

public class FileUploadHandler {

 private transient MultipartFile file;

 public void processFile() {
  //Do something with the MultipartFile here
 }

 public void setFile(MultipartFile file) {
  this.file = file;
 }
}
```

you can process the upload using a transition action as in the following example:

```
<view-state id="uploadFile" model="uploadFileHandler">
 <var name="fileUploadHandler" class="org.springframework.webflow.samples.booking.FileUploadHandler" />
 <transition on="upload" to="finish" >
  <evaluate expression="fileUploadHandler.processFile()"/>
 </transition>
 <transition on="cancel" to="finish" bind="false"/>
</view-state>
```

The `MultipartFile` will be bound to the `FileUploadHandler` bean as part of the normal form binding process so that it will be available to process during the execution of the transition action.

7. Flow Managed Persistence

7.1. Introduction

Most applications access data in some way. Many modify data shared by multiple users and therefore require transactional data access properties. They often transform relational data sets into domain objects to support application processing. Web Flow offers "flow managed persistence" where a flow can create, commit, and close a object persistence context for you. Web Flow integrates both Hibernate and JPA object persistence technologies.

Apart from flow-managed persistence, there is the pattern of fully encapsulating PersistenceContext management within the service layer of your application. In that case, the web layer does not get involved with persistence, instead it works entirely with detached objects that are passed to and returned by your service layer. This chapter will focus on the flow-managed persistence, exploring how and when to use this feature.

7.2. FlowScoped PersistenceContext

This pattern creates a `PersistenceContext` in `flowScope` on flow startup, uses that context for data access during the course of flow execution, and commits changes made to persistent entities at the end. This pattern provides isolation of intermediate edits by only committing changes to the database at the end of flow execution. This pattern is often used in conjunction with an optimistic locking strategy to protect the integrity of data modified in parallel by multiple users. To support saving and restarting the progress of a flow over an extended period of time, a durable store for flow state must be used. If a save and restart capability is not required, standard HTTP session-based storage of flow state is sufficient. In that case, session expiration or termination before commit could potentially result in changes being lost.

To use the FlowScoped PersistenceContext pattern, first mark your flow as a `persistence-context`:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow.xsd">

    <persistence-context />

</flow>
```

Then configure the correct `FlowExecutionListener` to apply this pattern to your flow. If using Hibernate, register the `HibernateFlowExecutionListener`. If using JPA, register the `JpaFlowExecutionListener`.

```
<webflow:flow-executor id="flowExecutor" flow-registry="flowRegistry">
    <webflow:flow-execution-listeners>
        <webflow:listener ref="jpaFlowExecutionListener" />
    </webflow:flow-execution-listeners>
</webflow:flow-executor>

<bean id="jpaFlowExecutionListener"
      class="org.springframework.webflow.persistence.JpaFlowExecutionListener">
    <constructor-arg ref="entityManagerFactory" />
    <constructor-arg ref="transactionManager" />
</bean>
```

To trigger a commit at the end, annotate your end-state with the commit attribute:

```
<end-state id="bookingConfirmed" commit="true" />
```

That is it. When your flow starts, the listener will handle allocating a new `EntityManager` in `flowScope`. Reference this EntityManager at anytime from within your flow by using the special `persistenceContext` variable. In addition, any data access that occurs using a Spring managed data access object will use this EntityManager automatically. Such data access operations should always execute non transactionally or in read-only transactions to maintain isolation of intermediate edits.

7.3. Flow Managed Persistence And Sub-Flows

A flow managed `PersistenceContext` is automatically extended (propagated) to subflows assuming the subflow also has the `<perstistence-context/>` variable. When a subflow re-uses the `PersistenceContext` started by its parent it ignores commit flags when an end state is reached thereby deferring the final decision (to commit or not) to its parent.

8. Securing Flows

8.1. Introduction

Security is an important concept for any application. End users should not be able to access any portion of a site simply by guessing the URL. Areas of a site that are sensitive must ensure that only authorized requests are processed. Spring Security is a proven security platform that can integrate with your application at multiple levels. This section will focus on securing flow execution.

8.2. How do I secure a flow?

Securing flow execution is a three step process:

- Configure Spring Security with authentication and authorization rules
- Annotate the flow definition with the secured element to define the security rules
- Add the SecurityFlowExecutionListener to process the security rules.

Each of these steps must be completed or else flow security rules will not be applied.

8.3. The secured element

The secured element designates that its containing element should apply the authorization check before fully entering. This may not occur more then once per stage of the flow execution that is secured.

Three phases of flow execution can be secured: flows, states and transitions. In each case the syntax for the secured element is identical. The secured element is located inside the element it is securing. For example, to secure a state the secured element occurs directly inside that state:

```
<view-state id="secured-view">
    <secured attributes="ROLE_USER" />
    ...
</view-state>
```

Security attributes

The `attributes` attribute is a comma separated list of Spring Security authorization attributes. Often, these are specific security roles. The attributes are compared against the user's granted attributes by a Spring Security access decision manager.

```
<secured attributes="ROLE_USER" />
```

By default, a role based access decision manager is used to determine if the user is allowed access. This will need to be overridden if your application is not using authorization roles.

Matching type

There are two types of matching available: `any` and `all`. Any, allows access if at least one of the required security attributes is granted to the user. All, allows access only if each of the required security attributes are granted to the user.

```
<secured attributes="ROLE_USER, ROLE_ANONYMOUS" match="any" />
```

This attribute is optional. If not defined, the default value is `any`.

The `match` attribute will only be respected if the default access decision manager is used.

8.4. The SecurityFlowExecutionListener

Defining security rules in the flow by themselves will not protect the flow execution. A `SecurityFlowExecutionListener` must also be defined in the webflow configuration and applied to the flow executor.

```
<webflow:flow-executor id="flowExecutor" flow-registry="flowRegistry">
    <webflow:flow-execution-listeners>
        <webflow:listener ref="securityFlowExecutionListener" />
    </webflow:flow-execution-listeners>
</webflow:flow-executor>

<bean id="securityFlowExecutionListener"
      class="org.springframework.webflow.security.SecurityFlowExecutionListener" />
```

If access is denied to a portion of the application an `AccessDeniedException` will be thrown. This exception will later be caught by Spring Security and used to prompt the user to authenticate. It is important that this exception be allowed to travel up the execution stack uninhibited, otherwise the end user may not be prompted to authenticate.

Custom Access Decision Managers

If your application is using authorities that are not role based, you will need to configure a custom `AccessDecisionManager`. You can override the default decision manager by setting the `accessDecisionManager` property on the security listener. Please consult the Spring Security reference documentation to learn more about decision managers.

```
<bean id="securityFlowExecutionListener"
      class="org.springframework.webflow.security.SecurityFlowExecutionListener">
    <property name="accessDecisionManager" ref="myCustomAccessDecisionManager" />
</bean>
```

8.5. Configuring Spring Security

Spring Security has robust configuration options available. As every application and environment has its own security requirements, the Spring Security reference documentation is the best place to learn the available options.

Both the `booking-faces` and `booking-mvc` sample applications are configured to use Spring Security. Configuration is needed at both the Spring and web.xml levels.

Spring configuration

The Spring configuration defines `http` specifics (such as protected URLs and login/logout mechanics) and the `authentication-provider`. For the sample applications, a local authentication provider is configured.

```
<security:http auto-config="true">
    <security:form-login login-page="/spring/login"
                         login-processing-url="/spring/loginProcess"
                         default-target-url="/spring/main"
                         authentication-failure-url="/spring/login?login_error=1" />
    <security:logout logout-url="/spring/logout" logout-success-url="/spring/logout-success" />
</security:http>

<security:authentication-provider>
    <security:password-encoder hash="md5" />
    <security:user-service>
        <security:user name="keith" password="417c7382b16c395bc25b5da1398cf076"
                       authorities="ROLE_USER,ROLE_SUPERVISOR" />
        <security:user name="erwin" password="12430911a8af075c6f41c6976af22b09"
                       authorities="ROLE_USER,ROLE_SUPERVISOR" />
        <security:user name="jeremy" password="57c6cbff0d421449be820763f03139eb"
                       authorities="ROLE_USER" />
        <security:user name="scott" password="942f2339bf50796de535a384f0d1af3e"
                       authorities="ROLE_USER" />
    </security:user-service>
</security:authentication-provider>
```

web.xml Configuration

In the `web.xml` file, a `filter` is defined to intercept all requests. This filter will listen for login/logout requests and process them accordingly. It will also catch `AccesDeniedExceptions` and redirect the user to the login page.

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

9. Flow Inheritance

9.1. Introduction

Flow inheritance allows one flow to inherit the configuration of another flow. Inheritance can occur at both the flow and state levels. A common use case is for a parent flow to define global transitions and exception handlers, then each child flow can inherit those settings.

In order for a parent flow to be found, it must be added to the `flow-registry` just like any other flow.

9.2. Is flow inheritance like Java inheritance?

Flow inheritance is similar to Java inheritance in that elements defined in a parent are exposed via the child, however, there are key differences.

A child flow cannot override an element from a parent flow. Similar elements between the parent and child flows will be merged. Unique elements in the parent flow will be added to the child.

A child flow can inherit from multiple parent flows. Java inheritance is limited to a single class.

9.3. Types of Flow Inheritance

Flow level inheritance

Flow level inheritance is defined by the `parent` attribute on the `flow` element. The attribute contains a comma separated list of flow identifiers to inherit from. The child flow will inherit from each parent in the order it is listed adding elements and content to the resulting flow. The resulting flow from the first merge will be considered the child in the second merge, and so on.

```
<flow parent="common-transitions, common-states">
```

State level inheritance

State level inheritance is similar to flow level inheritance, except only one state inherits from the parent, instead of the entire flow.

Unlike flow inheritance, only a single parent is allowed. Additionally, the identifier of the flow state to inherit from must also be defined. The identifiers for the flow and the state within that flow are separated by a #.

The parent and child states must be of the same type. For instance a view-state cannot inherit from an end-state, only another view-state.

```
<view-state id="child-state" parent="parent-flow#parent-view-state">
```

Note

The intent for flow-level inheritance is to define common states to be added to and shared among multiple flow definitions while the intent for state-level inheritance is to extend from and merge with a single parent state. Flow-level inheritance is a good fit for composition and multiple inheritance but at the state level you can still only inherit from a single parent state.

9.4. Abstract flows

Often parent flows are not designed to be executed directly. In order to protect these flows from running, they can be marked as `abstract`. If an abstract flow attempts to run, a `FlowBuilderException` will be thrown.

```
<flow abstract="true">
```

9.5. Inheritance Algorithm

When a child flow inherits from it's parent, essentially what happens is that the parent and child are merged together to create a new flow. There are rules for every element in the Web Flow definition language that govern how that particular element is merged.

There are two types of elements: *mergeable* and *non-mergeable*. Mergeable elements will always attempt to merge together if the elements are similar. Non-mergeable elements in a parent or child flow will always be contained in the resulting flow intact. They will not be modified as part of the merge process.

Note

Paths to external resources in the parent flow should be absolute. Relative paths will break when the two flows are merged unless the parent and child flow are in the same directory. Once merged, all relative paths in the parent flow will become relative to the child flow.

Mergeable Elements

If the elements are of the same type and their keyed attribute are identical, the content of the parent element will be merged with the child element. The merge algorithm will continue to merge each sub-element of the merging parent and child. Otherwise the parent element is added as a new element to the child.

In most cases, elements from a parent flow that are added will be added after elements in the child flow. Exceptions to this rule include action elements (evaluate, render and set) which will be added at the beginning. This allows for the results of parent actions to be used by child actions.

Mergeable elements are:

- action-state: id
- attribute: name
- decision-state: id
- end-state: id
- flow: always merges
- if: test

- on-end: always merges
- on-entry: always merges
- on-exit: always merges
- on-render: always merges
- on-start: always merges
- input: name
- output: name
- secured: attributes
- subflow-state: id
- transition: on and on-exception
- view-state: id

Non-mergeable Elements

Non-mergeable elements are:

- bean-import
- evaluate
- exception-handler
- persistence-context
- render
- set
- var

10. System Setup

10.1. Introduction

This chapter shows you how to setup the Web Flow system for use in any web environment.

10.2. Java Config and XML Namespace

Web Flow provides dedicated configuration support for both Java and XML-based configuration.

To get started with XML based configuration declare the webflow config XML namespace:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:webflow="http://www.springframework.org/schema/webflow-config"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/webflow-config
           http://www.springframework.org/schema/webflow-config/spring-webflow-config.xsd">

    <!-- Setup Web Flow here -->

</beans>
```

To get started with Java configuration extend `AbstractFlowConfiguration` in an `@Configuration` class:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.webflow.config.AbstractFlowConfiguration;

@Configuration
public class WebFlowConfig extends AbstractFlowConfiguration {

}
```

10.3. Basic system configuration

The next section shows the minimal configuration required to set up the Web Flow system in your application.

FlowRegistry

Register your flows in a `FlowRegistry` in XML:

```
<webflow:flow-registry id="flowRegistry">
    <webflow:flow-location path="/WEB-INF/flows/booking/booking.xml" />
</webflow:flow-registry>
```

Register your flows in a `FlowRegistry` in Java:

```
@Bean
public FlowDefinitionRegistry flowRegistry() {
    return getFlowDefinitionRegistryBuilder()
        .addFlowLocation("/WEB-INF/flows/booking/booking.xml")
        .build();
}
```

FlowExecutor

Deploy a FlowExecutor, the central service for executing flows in XML:

```
<webflow:flow-executor id="flowExecutor" />
```

Deploy a FlowExecutor, the central service for executing flows in Java:

```
@Bean
public FlowExecutor flowExecutor() {
    return getFlowExecutorBuilder(flowRegistry()).build();
}
```

See the Spring MVC and Spring Faces sections of this guide on how to integrate the Web Flow system with the MVC and JSF environment, respectively.

10.4. flow-registry options

This section explores flow-registry configuration options.

Specifying flow locations

Use the `location` element to specify paths to flow definitions to register. By default, flows will be assigned registry identifiers equal to their filenames minus the file extension, unless a registry bath path is defined.

In XML:

```
<webflow:flow-location path="/WEB-INF/flows/booking/booking.xml" />
```

In Java:

```
return getFlowDefinitionRegistryBuilder()
        .addFlowLocation("/WEB-INF/flows/booking/booking.xml")
        .build();
```

Assigning custom flow identifiers

Specify an id to assign a custom registry identifier to a flow in XML:

```
<webflow:flow-location path="/WEB-INF/flows/booking/booking.xml" id="bookHotel" />
```

Specify an id to assign a custom registry identifier to a flow in Java:

```
return getFlowDefinitionRegistryBuilder()
        .addFlowLocation("/WEB-INF/flows/booking/booking.xml", "bookHotel")
        .build();
```

Assigning flow meta-attributes

Use the `flow-definition-attributes` element to assign custom meta-attributes to a registered flow.

In XML:

```
<webflow:flow-location path="/WEB-INF/flows/booking/booking.xml">
    <webflow:flow-definition-attributes>
        <webflow:attribute name="caption" value="Books a hotel" />
    </webflow:flow-definition-attributes>
</webflow:flow-location>
```

In Java:

```
Map<String, Object> attrs = ... ;

return getFlowDefinitionRegistryBuilder()
        .addFlowLocation("/WEB-INF/flows/booking/booking.xml", null, attrs)
        .build();
```

Registering flows using a location pattern

Use the `flow-location-patterns` element to register flows that match a specific resource location pattern:

In XML:

```
<webflow:flow-location-pattern value="/WEB-INF/flows/**/*-flow.xml" />
```

In Java:

```
return getFlowDefinitionRegistryBuilder()
        .addFlowLocationPattern("/WEB-INF/flows/**/*-flow.xml")
        .build();
```

Flow location base path

Use the `base-path` attribute to define a base location for all flows in the application. All flow locations are then relative to the base path. The base path can be a resource path such as '/WEB-INF' or a location on the classpath like 'classpath:org/springframework/webflow/samples'.

In XML:

```
<webflow:flow-registry id="flowRegistry" base-path="/WEB-INF">
    <webflow:flow-location path="/hotels/booking/booking.xml" />
</webflow:flow-registry>
```

In Java:

```
return getFlowDefinitionRegistryBuilder()
        .setBasePath("/WEB-INF")
        .addFlowLocationPattern("/hotels/booking/booking.xml")
        .build();
```

With a base path defined, the algorithm that assigns flow identifiers changes slightly. Flows will now be assigned registry identifiers equal to the the path segment between their base path and file name. For example, if a flow definition is located at '/WEB-INF/hotels/booking/booking-flow.xml' and the base path is '/WEB-INF' the remaining path to this flow is 'hotels/booking' which becomes the flow id.

> **Directory per flow definition**
>
> Recall it is a best practice to package each flow definition in a unique directory. This improves modularity, allowing dependent resources to be packaged with the flow definition. It also prevents two flows from having the same identifiers when using the convention.

If no base path is not specified or if the flow definition is directly on the base path, flow id assignment from the filename (minus the extension) is used. For example, if a flow definition file is 'booking.xml', the flow identifier is simply 'booking'.

Location patterns are particularly powerful when combined with a registry base path. Instead of the flow identifiers becoming '*-flow', they will be based on the directory path. For example in XML:

```
<webflow:flow-registry id="flowRegistry" base-path="/WEB-INF">
    <webflow:flow-location-pattern value="/**/*-flow.xml" />
</webflow:flow-registry>
```

In Java:

```
return getFlowDefinitionRegistryBuilder()
        .setBasePath("/WEB-INF")
        .addFlowLocationPattern("/**/*-flow.xml")
        .build();
```

In the above example, suppose you had flows located in `/user/login`, `/user/registration`, `/hotels/booking`, and `/flights/booking` directories within `WEB-INF`, you'd end up with flow ids of `user/login`, `user/registration`, `hotels/booking`, and `flights/booking`, respectively.

Configuring FlowRegistry hierarchies

Use the `parent` attribute to link two flow registries together in a hierarchy. When the child registry is queried, if it cannot find the requested flow it will delegate to its parent.

In XML:

```
<!-- my-system-config.xml -->
<webflow:flow-registry id="flowRegistry" parent="sharedFlowRegistry">
    <webflow:flow-location path="/WEB-INF/flows/booking/booking.xml" />
</webflow:flow-registry>

<!-- shared-config.xml -->
<webflow:flow-registry id="sharedFlowRegistry">
    <!-- Global flows shared by several applications -->
</webflow:flow-registry>
```

In Java:

```
@Configuration
public class WebFlowConfig extends AbstractFlowConfiguration {

    @Autowired
    private SharedConfig sharedConfig;

    @Bean
    public FlowDefinitionRegistry flowRegistry() {
        return getFlowDefinitionRegistryBuilder()
                .setParent(this.sharedConfig.sharedFlowRegistry())
                .addFlowLocation("/WEB-INF/flows/booking/booking.xml")
                .build();
    }
}

@Configuration
public class SharedConfig extends AbstractFlowConfiguration {

    @Bean
    public FlowDefinitionRegistry sharedFlowRegistry() {
        return getFlowDefinitionRegistryBuilder()
                .addFlowLocation("/WEB-INF/flows/shared.xml")
                .build();
    }
}
```

Configuring custom FlowBuilder services

Use the `flow-builder-services` attribute to customize the services and settings used to build flows in a flow-registry. If no flow-builder-services tag is specified, the default service implementations are used. When the tag is defined, you only need to reference the services you want to customize.

In XML:

```
<webflow:flow-registry id="flowRegistry" flow-builder-services="flowBuilderServices">
    <webflow:flow-location path="/WEB-INF/flows/booking/booking.xml" />
</webflow:flow-registry>

<webflow:flow-builder-services id="flowBuilderServices" />
```

In Java:

```
@Bean
public FlowDefinitionRegistry flowRegistry() {
 return getFlowDefinitionRegistryBuilder(flowBuilderServices())
            .addFlowLocation("/WEB-INF/flows/booking/booking.xml")
            .build();
}

@Bean
public FlowBuilderServices flowBuilderServices() {
    return getFlowBuilderServicesBuilder().build();
}
```

The configurable services are the `conversion-service`, `expression-parser`, and `view-factory-creator`. These services are configured by referencing custom beans you define.

For example in XML:

```
<webflow:flow-builder-services id="flowBuilderServices"
    conversion-service="conversionService"
    expression-parser="expressionParser"
    view-factory-creator="viewFactoryCreator" />

<bean id="conversionService" class="..." />
<bean id="expressionParser" class="..." />
<bean id="viewFactoryCreator" class="..." />
```

In Java:

```
@Bean
public FlowBuilderServices flowBuilderServices() {
    return getFlowBuilderServicesBuilder()
            .setConversionService(conversionService())
            .setExpressionParser(expressionParser)
            .setViewFactoryCreator(mvcViewFactoryCreator())
            .build();
}

@Bean
public ConversionService conversionService() {
    // ...
}

@Bean
public ExpressionParser expressionParser() {
    // ...
}

@Bean
public ViewFactoryCreator viewFactoryCreator() {
    // ...
}
```

conversion-service

Use the `conversion-service` attribute to customize the `ConversionService` used by the Web Flow system. Type conversion is used to convert from one type to another when required during flow execution such as when processing request parameters, invoking actions, and so on. Many common object types such as numbers, classes, and enums are supported. However you'll probably need to provide your own type conversion and formatting logic for custom data types. Please read Section 5.7, “Performing type conversion” for important information on how to provide custom type conversion logic.

expression-parser

Use the `expression-parser` attribute to customize the `ExpressionParser` used by the Web Flow system. The default ExpressionParser uses the Unified EL if available on the classpath, otherwise Spring EL is used.

view-factory-creator

Use the `view-factory-creator` attribute to customize the `ViewFactoryCreator` used by the Web Flow system. The default ViewFactoryCreator produces Spring MVC ViewFactories capable of rendering JSP, Velocity, and Freemarker views.

The configurable settings are `development`. These settings are global configuration attributes that can be applied during the flow construction process.

development

Set this to `true` to switch on flow *development mode*. Development mode switches on hot-reloading of flow definition changes, including changes to dependent flow resources such as message bundles.

10.5. flow-executor options

This section explores flow-executor configuration options.

Attaching flow execution listeners

Use the `flow-execution-listeners` element to register listeners that observe the lifecycle of flow executions. For example in XML:

```
<webflow:flow-execution-listeners>
    <webflow:listener ref="securityListener"/>
    <webflow:listener ref="persistenceListener"/>
</webflow:flow-execution-listeners>
```

In Java:

```
@Bean
public FlowExecutor flowExecutor() {
    return getFlowExecutorBuilder(flowRegistry())
            .addFlowExecutionListener(securityListener())
            .addFlowExecutionListener(persistenceListener())
            .build();
}
```

You may also configure a listener to observe only certain flows. For example in XML:

```
<webflow:listener ref="securityListener" criteria="securedFlow1,securedFlow2"/>
```

In Java:

```
@Bean
public FlowExecutor flowExecutor() {
    return getFlowExecutorBuilder(flowRegistry())
            .addFlowExecutionListener(securityListener(), "securedFlow1,securedFlow2")
            .build();
}
```

Tuning FlowExecution persistence

Use the `flow-execution-repository` element to tune flow execution persistence settings. For example in XML:

```
<webflow:flow-executor id="flowExecutor" flow-registry="flowRegistry">
    <webflow:flow-execution-repository max-executions="5" max-execution-snapshots="30" />
</webflow:flow-executor>
```

In Java:

```
@Bean
public FlowExecutor flowExecutor() {
    return getFlowExecutorBuilder(flowRegistry())
            .setMaxFlowExecutions(5)
            .setMaxFlowExecutionSnapshots(30)
            .build();
}
```

max-executions

Tune the `max-executions` attribute to place a cap on the number of flow executions that can be created per user session. When the maximum number of executions is exceeded, the oldest execution is removed.

Note

The `max-executions` attribute is per user session, i.e. it works across instances of any flow definition.

max-execution-snapshots

Tune the `max-execution-snapshots` attribute to place a cap on the number of history snapshots that can be taken per flow execution. To disable snapshotting, set this value to 0. To enable an unlimited number of snapshots, set this value to -1.

Note

History snapshots enable browser back button support. When snapshotting is disabled pressing the browser back button will not work. It will result in using an execution key that points to a snapshot that has not be recorded.

11. Spring MVC Integration

11.1. Introduction

This chapter shows how to integrate Web Flow into a Spring MVC web application. The `booking-mvc` sample application is a good reference for Spring MVC with Web Flow. This application is a simplified travel site that allows users to search for and book hotel rooms.

11.2. Configuring web.xml

The first step to using Spring MVC is to configure the `DispatcherServlet` in `web.xml`. You typically do this once per web application.

The example below maps all requests that begin with `/spring/` to the DispatcherServlet. An `init-param` is used to provide the `contextConfigLocation`. This is the configuration file for the web application.

```
<servlet>
 <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
 <init-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/web-application-config.xml</param-value>
 </init-param>
</servlet>

<servlet-mapping>
 <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
 <url-pattern>/spring/*</url-pattern>
</servlet-mapping>
```

11.3. Dispatching to flows

The `DispatcherServlet` maps requests for application resources to handlers. A flow is one type of handler.

Registering the FlowHandlerAdapter

The first step to dispatching requests to flows is to enable flow handling within Spring MVC. To this, install the `FlowHandlerAdapter`:

```
<!-- Enables FlowHandler URL mapping -->
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
 <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

Defining flow mappings

Once flow handling is enabled, the next step is to map specific application resources to your flows. The simplest way to do this is to define a `FlowHandlerMapping`:

```xml
<!-- Maps request paths to flows in the flowRegistry;
 e.g. a path of /hotels/booking looks for a flow with id "hotels/booking" -->
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
 <property name="flowRegistry" ref="flowRegistry"/>
 <property name="order" value="0"/>
</bean>
```

Configuring this mapping allows the Dispatcher to map application resource paths to flows in a flow registry. For example, accessing the resource path `/hotels/booking` would result in a registry query for the flow with id `hotels/booking`. If a flow is found with that id, that flow will handle the request. If no flow is found, the next handler mapping in the Dispatcher's ordered chain will be queried or a "noHandlerFound" response will be returned.

Flow handling workflow

When a valid flow mapping is found, the `FlowHandlerAdapter` figures out whether to start a new execution of that flow or resume an existing execution based on information present the HTTP request. There are a number of defaults related to starting and resuming flow executions the adapter employs:

- HTTP request parameters are made available in the input map of all starting flow executions.
- When a flow execution ends without sending a final response, the default handler will attempt to start a new execution in the same request.
- Unhandled exceptions are propagated to the Dispatcher unless the exception is a NoSuchFlowExecutionException. The default handler will attempt to recover from a NoSuchFlowExecutionException by starting over a new execution.

Consult the API documentation for `FlowHandlerAdapter` for more information. You may override these defaults by subclassing or by implementing your own FlowHandler, discussed in the next section.

11.4. Implementing custom FlowHandlers

`FlowHandler` is the extension point that can be used to customize how flows are executed in a HTTP servlet environment. A `FlowHandler` is used by the `FlowHandlerAdapter` and is responsible for:

- Returning the `id` of a flow definition to execute
- Creating the input to pass new executions of that flow as they are started
- Handling outcomes returned by executions of that flow as they end
- Handling any exceptions thrown by executions of that flow as they occur

These responsibilities are illustrated in the definition of the `org.springframework.mvc.servlet.FlowHandler` interface:

```
public interface FlowHandler {

 public String getFlowId();

 public MutableAttributeMap createExecutionInputMap(HttpServletRequest request);

 public String handleExecutionOutcome(FlowExecutionOutcome outcome,
  HttpServletRequest request, HttpServletResponse response);

 public String handleException(FlowException e,
  HttpServletRequest request, HttpServletResponse response);
}
```

To implement a FlowHandler, subclass `AbstractFlowHandler`. All these operations are optional, and if not implemented the defaults will apply. You only need to override the methods that you need. Specifically:

- Override `getFlowId(HttpServletRequest)` when the id of your flow cannot be directly derived from the HTTP request. By default, the id of the flow to execute is derived from the pathInfo portion of the request URI. For example, `http://localhost/app/hotels/booking?hotelId=1` results in a flow id of `hotels/booking` by default.
- Override `createExecutionInputMap(HttpServletRequest)` when you need fine-grained control over extracting flow input parameters from the HttpServletRequest. By default, all request parameters are treated as flow input parameters.
- Override `handleExecutionOutcome` when you need to handle specific flow execution outcomes in a custom manner. The default behavior sends a redirect to the ended flow's URL to restart a new execution of the flow.
- Override `handleException` when you need fine-grained control over unhandled flow exceptions. The default behavior attempts to restart the flow when a client attempts to access an ended or expired flow execution. Any other exception is rethrown to the Spring MVC ExceptionResolver infrastructure by default.

Example FlowHandler

A common interaction pattern between Spring MVC And Web Flow is for a Flow to redirect to a @Controller when it ends. FlowHandlers allow this to be done without coupling the flow definition itself with a specific controller URL. An example FlowHandler that redirects to a Spring MVC Controller is shown below:

```
public class BookingFlowHandler extends AbstractFlowHandler {
 public String handleExecutionOutcome(FlowExecutionOutcome outcome,
          HttpServletRequest request, HttpServletResponse response) {
  if (outcome.getId().equals("bookingConfirmed")) {
   return "/booking/show?bookingId=" + outcome.getOutput().get("bookingId");
  } else {
   return "/hotels/index";
  }
 }
}
```

Since this handler only needs to handle flow execution outcomes in a custom manner, nothing else is overridden. The `bookingConfirmed` outcome will result in a redirect to show the new booking. Any other outcome will redirect back to the hotels index page.

Deploying a custom FlowHandler

To install a custom FlowHandler, simply deploy it as a bean. The bean name must match the id of the flow the handler should apply to.

```
<bean name="hotels/booking" class="org.springframework.webflow.samples.booking.BookingFlowHandler" />
```

With this configuration, accessing the resource `/hotels/booking` will launch the `hotels/booking` flow using the custom BookingFlowHandler. When the booking flow ends, the FlowHandler will process the flow execution outcome and redirect to the appropriate controller.

FlowHandler Redirects

A FlowHandler handling a FlowExecutionOutcome or FlowException returns a `String` to indicate the resource to redirect to after handling. In the previous example, the `BookingFlowHandler` redirects to the `booking/show` resource URI for `bookingConfirmed` outcomes, and the `hotels/index` resource URI for all other outcomes.

By default, returned resource locations are relative to the current servlet mapping. This allows for a flow handler to redirect to other Controllers in the application using relative paths. In addition, explicit redirect prefixes are supported for cases where more control is needed.

The explicit redirect prefixes supported are:

- `servletRelative:` - redirect to a resource relative to the current servlet
- `contextRelative:` - redirect to a resource relative to the current web application context path
- `serverRelative:` - redirect to a resource relative to the server root
- `http://` or `https://` - redirect to a fully-qualified resource URI

These same redirect prefixes are also supported within a flow definition when using the `externalRedirect:` directive in conjunction with a view-state or end-state; for example, `view="externalRedirect:http://springframework.org"`

11.5. View Resolution

Web Flow 2 maps selected view identifiers to files located within the flow's working directory unless otherwise specified. For existing Spring MVC + Web Flow applications, an external `ViewResolver` is likely already handling this mapping for you. Therefore, to continue using that resolver and to avoid having to change how your existing flow views are packaged, configure Web Flow as follows:

```
<webflow:flow-registry id="flowRegistry" flow-builder-services="flowBuilderServices">
 <webflow:location path="/WEB-INF/hotels/booking/booking.xml" />
</webflow:flow-registry>

<webflow:flow-builder-services id="flowBuilderServices" view-factory-creator="mvcViewFactoryCreator"/>

<bean id="mvcViewFactoryCreator" class="org.springframework.webflow.mvc.builder.MvcViewFactoryCreator">
 <property name="viewResolvers" ref="myExistingViewResolverToUseForFlows"/>
</bean>
```

The MvcViewFactoryCreator is the factory that allows you to configure how the Spring MVC view system is used inside Spring Web Flow. Use it to configure existing ViewResolvers, as well as other services such as a custom MessageCodesResolver. You may also enable data binding use Spring MVC's native BeanWrapper by setting the `useSpringBinding` flag to true. This is an alternative to using the Unified EL for view-to-model data binding. See the JavaDoc API of this class for more information.

11.6. Signaling an event from a View

When a flow enters a view-state it pauses, redirects the user to its execution URL, and waits for a user event to resume. Events are generally signaled by activating buttons, links, or other user interface commands. How events are decoded server-side is specific to the view technology in use. This section shows how to trigger events from HTML-based views generated by templating engines such as JSP, Velocity, or Freemarker.

Using a named HTML button to signal an event

The example below shows two buttons on the same form that signal `proceed` and `cancel` events when clicked, respectively.

```
<input type="submit" name="_eventId_proceed" value="Proceed" />
<input type="submit" name="_eventId_cancel" value="Cancel" />
```

When a button is pressed Web Flow finds a request parameter name beginning with `_eventId_` and treats the remaining substring as the event id. So in this example, submitting `_eventId_proceed` becomes `proceed`. This style should be considered when there are several different events that can be signaled from the same form.

Using a hidden HTML form parameter to signal an event

The example below shows a form that signals the `proceed` event when submitted:

```
<input type="submit" value="Proceed" />
<input type="hidden" name="_eventId" value="proceed" />
```

Here, Web Flow simply detects the special `_eventId` parameter and uses its value as the event id. This style should only be considered when there is one event that can be signaled on the form.

Using a HTML link to signal an event

The example below shows a link that signals the `cancel` event when activated:

```
<a href="${flowExecutionUrl}&_eventId=cancel">Cancel</a>
```

Firing an event results in a HTTP request being sent back to the server. On the server-side, the flow handles decoding the event from within its current view-state. How this decoding process works is specific to the view implementation. Recall a Spring MVC view implementation simply looks for a request parameter named `_eventId`. If no `_eventId` parameter is found, the view will look for a parameter that starts with `_eventId_` and will use the remaining substring as the event id. If neither cases exist, no flow event is triggered.

11.7. Embedding A Flow On A Page

By default when a flow enters a view state, it executes a client-side redirect before rendering the view. This approach is known as POST-REDIRECT-GET. It has the advantage of separating the form processing for one view from the rendering of the next view. As a result the browser Back and Refresh buttons work seamlessly without causing any browser warnings.

Normally the client-side redirect is transparent from a user's perspective. However, there are situations where POST-REDIRECT-GET may not bring the same benefits. For example a flow may be embedded on a page and driven via Ajax requests refreshing only the area of the page that belongs to the flow. Not only is it unnecessary to use client-side redirects in this case, it is also not the desired behavior with regards to keeping the surrounding content of the page intact.

The Section 12.5, “Handling Ajax Requests” explains how to do partial rendering during Ajax requests. The focus of this section is to explain how to control flow execution redirect behavior during Ajax requests. To indicate a flow should execute in "page embedded" mode all you need to do is append an extra parameter when launching the flow:

```
/hotels/booking?mode=embedded
```

When launched in "page embedded" mode a flow will not issue flow execution redirects during Ajax requests. The mode=embedded parameter only needs to be passed when launching the flow. Your only other concern is to use Ajax requests and to render only the content required to update the portion of the page displaying the flow.

Embedded Mode Vs Default Redirect Behavior

By default Web Flow does a client-side redirect upon entering every view state. However if you remain in the same view state -- for example a transition without a "to" attribute -- during an Ajax request there will not be a client-side redirect. This behavior should be quite familiar to Spring Web Flow 2 users. It is appropriate for a top-level flow that supports the browser back button while still taking advantage of Ajax and partial rendering for use cases where you remain in the same view such as form validation, paging trough search results, and others. However transitions to a new view state are always followed with a client-side redirect. That makes it impossible to embed a flow on a page or within a modal dialog and execute more than one view state without causing a full-page refresh. Hence if your use case requires embedding a flow you can launch it in "embedded" mode.

Embedded Flow Examples

If you'd like to see examples of a flow embedded on a page and within a modal dialog please refer to the webflow-showcase project. You can check out the source code locally, build it as you would a Maven project, and import it into Eclipse:

```
cd some-directory
svn co https://src.springframework.org/svn/spring-samples/webflow-showcase
cd webflow-showcase
mvn package
# import into Eclipse
```

11.8. Saving Flow Output to MVC Flash Scope

Flow output can be automatically saved to MVC flash scope when an `end-state` performs an internal redirect. This is particularly useful when displaying a summary screen at the end of a flow. For backwards compatibility this feature is disabled by default, to enable set `saveOutputToFlashScopeOnRedirect` on your `FlowHandlerAdapter` to `true`.

```
<!-- Enables FlowHandler URL mapping -->
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
 <property name="flowExecutor" ref="flowExecutor" />
 <property name="saveOutputToFlashScopeOnRedirect" value="true" />
</bean>
```

The following example will add `confirmationNumber` to the MVC flash scope before redirecting to the `summary` screen.

```
<end-state id="finish" view="externalRedirect:summary">
 <output name="confirmationNumber" value="booking.confirmationNumber" />
</end-state>
```

12. Spring JavaScript Quick Reference

12.1. Introduction

The *spring-js-resources* module is a legacy module that is no longer recommended for use but is provided still as an optional module for backwards compatibility. Its original aim is to provide a client-side programming model for progressively enhancing a web page with behavior and Ajax remoting.

Use of the Spring JS API is demonstrated in the samples repository.

12.2. Serving Javascript Resources

The Spring Framework provides a mechanism for serving static resources. See the Spring Framework documentation). With the new <mvc:resources> element resource requests (.js, .css) are handled by the`DispatcherSevlet`. Here is example configuration in XML (Java config is also available):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:mvc="http://www.springframework.org/schema/mvc"
 xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework.org/schema/
mvc/spring-mvc.xsd
  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
beans.xsd">

 <mvc:annotation-driven/>

 <mvc:resources mapping="/resources/**" location="/, classpath:/META-INF/web-resources/" />

 ...

</beans>
```

This incoming maps requests for `/resources` to resources found under `/META-INF/web-resources` on the classpath. That's where Spring JavaScript resources are bundled. However, you can modify the location attribute in the above configuration in order to serve resources from any classpath or web application relative location.

Note that the full resource URL depends on how your DispatcherServlet is mapped. In the mvc-booking sample we've chosen to map it with the default servlet mapping '/':

```
<servlet>
 <servlet-name>DispatcherServlet</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>

<servlet-mapping>
 <servlet-name>DispatcherServlet</servlet-name>
 <url-pattern>/</url-pattern>
</servlet-mapping>
```

That means the full URL to load `Spring.js` is `/myapp/resources/spring/Spring.js`. If your `DispatcherServlet` was instead mapped to `/main/*` then the full URL would be `/myapp/main/resources/spring/Spring.js`.

When using of the default servlet mapping it is also recommended to add this to your Spring MVC configuration, which ensures that any resource requests not handled by your Spring MVC mappings will be delegated back to the Servlet container.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:mvc="http://www.springframework.org/schema/mvc"
 xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework.org/schema/
mvc/spring-mvc.xsd
  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
beans.xsd">

 ...

 <mvc:default-servlet-handler />

</beans>
```

12.3. Including Spring Javascript in a Page

Spring JS is designed such that an implementation of its API can be built for any of the popular Javascript toolkits. The initial implementation of Spring.js builds on the Dojo toolkit.

Using Spring Javascript in a page requires including the underlying toolkit as normal, the `Spring.js` base interface file, and the `Spring-(library implementation).js` file for the underlying toolkit. As an example, the following includes obtain the Dojo implementation of Spring.js using the `ResourceServlet`:

```
<script type="text/javascript" src="<c:url value="/resources/dojo/dojo.js" />"> </script>
<script type="text/javascript" src="<c:url value="/resources/spring/Spring.js" />"> </script>
<script type="text/javascript" src="<c:url value="/resources/spring/Spring-Dojo.js" />"> </script>
```

When using the widget system of an underlying library, typically you must also include some CSS resources to obtain the desired look and feel. For the booking-mvc reference application, Dojo's `tundra.css` is included:

```
<link type="text/css" rel="stylesheet" href="<c:url value="/resources/dijit/themes/tundra/tundra.css" /
>" />
```

12.4. Spring Javascript Decorations

A central concept in Spring Javascript is the notion of applying decorations to existing DOM nodes. This technique is used to progressively enhance a web page such that the page will still be functional in a less capable browser. The `addDecoration` method is used to apply decorations.

The following example illustrates enhancing a Spring MVC `<form:input>` tag with rich suggestion behavior:

```
<form:input id="searchString" path="searchString"/>
<script type="text/javascript">
 Spring.addDecoration(new Spring.ElementDecoration({
  elementId: "searchString",
  widgetType: "dijit.form.ValidationTextBox",
  widgetAttrs: { promptMessage : "Search hotels by name, address, city, or zip." }}));
</script>
```

The `ElementDecoration` is used to apply rich widget behavior to an existing DOM node. This decoration type does not aim to completely hide the underlying toolkit, so the toolkit's native widget type and attributes are used directly. This approach allows you to use a common decoration model to integrate any widget from the underlying toolkit in a consistent manner. See the `booking-mvc` reference application for more examples of applying decorations to do things from suggestions to client-side validation.

When using the `ElementDecoration` to apply widgets that have rich validation behavior, a common need is to prevent the form from being submitted to the server until validation passes. This can be done with the `ValidateAllDecoration`:

```
<input type="submit" id="proceed" name="_eventId_proceed" value="Proceed" />
<script type="text/javascript">
 Spring.addDecoration(new Spring.ValidateAllDecoration({ elementId:'proceed', event:'onclick' }));
</script>
```

This decorates the "Proceed" button with a special onclick event handler that fires the client side validators and does not allow the form to submit until they pass successfully.

An `AjaxEventDecoration` applies a client-side event listener that fires a remote Ajax request to the server. It also auto-registers a callback function to link in the response:

```
<a id="prevLink" href="search?searchString=${criteria.searchString}&page=${criteria.page -
 1}">Previous</a>
<script type="text/javascript">
 Spring.addDecoration(new Spring.AjaxEventDecoration({
  elementId: "prevLink",
  event: "onclick",
  params: { fragments: "body" }
 }));
</script>
```

This decorates the onclick event of the "Previous Results" link with an Ajax call, passing along a special parameter that specifies the fragment to be re-rendered in the response. Note that this link would still be fully functional if Javascript was unavailable in the client. (See Section 12.5, “Handling Ajax Requests” for details on how this request is handled on the server.)

It is also possible to apply more than one decoration to an element. The following example shows a button being decorated with Ajax and validate-all submit suppression:

```
<input type="submit" id="proceed" name="_eventId_proceed" value="Proceed" />
<script type="text/javascript">
 Spring.addDecoration(new Spring.ValidateAllDecoration({elementId:'proceed', event:'onclick'}));
 Spring.addDecoration(new Spring.AjaxEventDecoration({elementId:'proceed',
 event:'onclick',formId:'booking', params:{fragments:'messages'}}));
</script>
```

It is also possible to apply a decoration to multiple elements in a single statement using Dojo's query API. The following example decorates a set of checkbox elements as Dojo Checkbox widgets:

```
<div id="amenities">
<form:checkbox path="amenities" value="OCEAN_VIEW" label="Ocean View" /></li>
<form:checkbox path="amenities" value="LATE_CHECKOUT" label="Late Checkout" /></li>
<form:checkbox path="amenities" value="MINIBAR" label="Minibar" /></li>
<script type="text/javascript">
 dojo.query("#amenities input[type='checkbox']").forEach(function(element) {
  Spring.addDecoration(new Spring.ElementDecoration({
   elementId: element.id,
   widgetType : "dijit.form.CheckBox",
   widgetAttrs : { checked : element.checked }
  }));
 });
</script>
</div>
```

12.5. Handling Ajax Requests

Spring Javascript's client-side Ajax response handling is built upon the notion of receiving "fragments" back from the server. These fragments are just standard HTML that is meant to replace portions of the existing page. The key piece needed on the server is a way to determine which pieces of a full response need to be pulled out for partial rendering.

In order to be able to render partial fragments of a full response, the full response must be built using a templating technology that allows the use of composition for constructing the response, and for the member parts of the composition to be referenced and rendered individually. Spring Javascript provides some simple Spring MVC extensions that make use of Tiles to achieve this. The same technique could theoretically be used with any templating system supporting composition.

Spring Javascript's Ajax remoting functionality is built upon the notion that the core handling code for an Ajax request should not differ from a standard browser request, thus no special knowledge of an Ajax request is needed directly in the code and the same hanlder can be used for both styles of request.

Providing a Library-Specific AjaxHandler

The key interface for integrating various Ajax libraries with the Ajax-aware behavior of Web Flow (such as not redirecting for a partial page update) is `org.springframework.js.AjaxHandler`. A `SpringJavascriptAjaxHandler` is configured by default that is able to detect an Ajax request submitted via the Spring JS client-side API and can respond appropriately in the case where a redirect is required. In order to integrate a different Ajax library (be it a pure JavaScript library, or a higher-level abstraction such as an Ajax-capable JSF component library), a custom `AjaxHandler` can be injected into the `FlowHandlerAdapter` or `FlowController`.

Handling Ajax Requests with Spring MVC Controllers

In order to handle Ajax requests with Spring MVC controllers, all that is needed is the configuration of the provided Spring MVC extensions in your Spring application context for rendering the partial response (note that these extensions require the use of Tiles for templating):

```
<bean id="tilesViewResolver" class="org.springframework.webflow.mvc.view.AjaxUrlBasedViewResolver">
 <property name="viewClass" value="org.springframework.webflow.mvc.view.FlowAjaxTiles3View"/>
</bean>
```

This configures the `AjaxUrlBasedViewResolver` which in turn interprets Ajax requests and creates `FlowAjaxTilesView` objects to handle rendering of the appropriate fragments. Note that `FlowAjaxTilesView` is capable of handling the rendering for both Web Flow and pure Spring MVC requests. The fragments correspond to individual attributes of a Tiles view definition. For example, take the following Tiles view definition:

```
<definition name="hotels/index" extends="standardLayout">
 <put-attribute name="body" value="index.body" />
</definition>

<definition name="index.body" template="/WEB-INF/hotels/index.jsp">
 <put-attribute name="hotelSearchForm" value="/WEB-INF/hotels/hotelSearchForm.jsp" />
 <put-attribute name="bookingsTable" value="/WEB-INF/hotels/bookingsTable.jsp" />
</definition>
```

An Ajax request could specify the "body", "hotelSearchForm" or "bookingsTable" to be rendered as fragments in the request.

Handling Ajax Requests with Spring MVC + Spring Web Flow

Spring Web Flow handles the optional rendering of fragments directly in the flow definition language through use of the `render` element. The benefit of this approach is that the selection of fragments is completely decoupled from client-side code, such that no special parameters need to be passed with the request the way they currently must be with the pure Spring MVC controller approach. For example, if you wanted to render the "hotelSearchForm" fragment from the previous example Tiles view into a rich Javascript popup:

```
<view-state id="changeSearchCriteria" view="enterSearchCriteria.xhtml" popup="true">
 <on-entry>
  <render fragments="hotelSearchForm" />
 </on-entry>
 <transition on="search" to="reviewHotels">
  <evaluate expression="searchCriteria.resetPage()"/>
 </transition>
</view-state>
```

13. JSF Integration

13.1. Introduction

Spring Web Flow provides a JSF integration that lets you use the JSF UI Component Model with Spring Web Flow controllers. Web Flow also provides a Spring Security tag library for use in JSF environments, see Section 13.9, “Using the Spring Security Facelets Tag Library” for more details.

Spring Web Flow 2.5 requires JSF 2.2 or higher.

13.2. Configuring web.xml

The first step is to route requests to the `DispatcherServlet` in the `web.xml` file. In this example, we map all URLs that begin with `/spring/` to the servlet. The servlet needs to be configured. An `init-param` is used in the servlet to pass the `contextConfigLocation`. This is the location of the Spring configuration for your web application.

```
<servlet>
 <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
 <init-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/web-application-config.xml</param-value>
 </init-param>
 <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
 <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
 <url-pattern>/spring/*</url-pattern>
</servlet-mapping>
```

In order for JSF to bootstrap correctly, the `FacesServlet` must be configured in `web.xml` as it normally would even though you generally will not need to route requests through it at all when using JSF with Spring Web Flow.

```
<!-- Just here so the JSF implementation can initialize, *not* used at runtime -->
<servlet>
 <servlet-name>Faces Servlet</servlet-name>
 <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
 <load-on-startup>1</load-on-startup>
</servlet>

<!-- Just here so the JSF implementation can initialize -->
<servlet-mapping>
 <servlet-name>Faces Servlet</servlet-name>
 <url-pattern>*.faces</url-pattern>
</servlet-mapping>
```

The use of Facelets instead of JSP typically requires this in web.xml:

```
!-- Use JSF view templates saved as *.xhtml, for use with Facelets -->
<context-param>
 <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
 <param-value>.xhtml</param-value>
</context-param>
```

13.3. Configuring Web Flow for use with JSF

This section explains how to configure Web Flow with JSF. Both Java and XML style configuration are supported. The following is sample configuration for Web Flow and JSF in XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:webflow="http://www.springframework.org/schema/webflow-config"
 xmlns:faces="http://www.springframework.org/schema/faces"
 si:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/webflow-config
  http://www.springframework.org/schema/webflow-config/spring-webflow-config.xsd
  http://www.springframework.org/schema/faces
  http://www.springframework.org/schema/faces/spring-faces.xsd">

 <!-- Executes flows: the central entry point into the Spring Web Flow system -->
 <webflow:flow-executor id="flowExecutor">
  <webflow:flow-execution-listeners>
   <webflow:listener ref="facesContextListener"/>
  </webflow:flow-execution-listeners>
 </webflow:flow-executor>

 <!-- The registry of executable flow definitions -->
 <webflow:flow-registry id="flowRegistry" flow-builder-services="flowBuilderServices" base-path="/WEB-
INF">
  <webflow:flow-location-pattern value="**/*-flow.xml" />
 </webflow:flow-registry>

 <!-- Configures the Spring Web Flow JSF integration -->
 <faces:flow-builder-services id="flowBuilderServices" />

 <!-- A listener maintain one FacesContext instance per Web Flow request. -->
 <bean id="facesContextListener"
  class="org.springframework.faces.webflow.FlowFacesContextLifecycleListener" />

</beans>
```

The following is an example of the same in Java configuration:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.faces.config.*;

@Configuration
public class WebFlowConfig extends AbstractFacesFlowConfiguration {

    @Bean
    public FlowExecutor flowExecutor() {
        return getFlowExecutorBuilder(flowRegistry())
                .addFlowExecutionListener(new FlowFacesContextLifecycleListener())
                .build();
    }

    @Bean
    public FlowDefinitionRegistry flowRegistry() {
        return getFlowDefinitionRegistryBuilder()
                .setBasePath("/WEB-INF")
                .addFlowLocationPattern("**/*-flow.xml").build();

}
```

The main points are the installation of a `FlowFacesContextLifecycleListener` that manages a single FacesContext for the duration of Web Flow request and the use of the `flow-builder-services` element from the `faces` custom namespace to configure rendering for a JSF environment.

In a JSF environment you'll also need this Spring MVC related configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:faces="http://www.springframework.org/schema/faces"
   xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/faces
    http://www.springframework.org/schema/faces/spring-faces.xsd">

 <faces:resources />

 <bean class="org.springframework.faces.webflow.JsfFlowHandlerAdapter">
  <property name="flowExecutor" ref="flowExecutor" />
 </bean>

</beans>
```

The `resources` custom namespace element delegates JSF resource requests to the JSF resource API. The `JsfFlowHandlerAdapter` is a replacement for the `FlowHandlerAdapter` normally used with Web Flow. This adapter initializes itself with a `JsfAjaxHandler` instead of the `SpringJavaSciprtAjaxHandler`.

When using Java config, the `AbstractFacesFlowConfiguration` base class automatically registers `JsfResourceRequestHandler` so there is nothing further to do.

13.4. Replacing the JSF Managed Bean Facility

When using JSF with Spring Web Flow you can completely replace the JSF managed bean facility with a combination of Web Flow managed variables and Spring managed beans. It gives you a good deal more control over the lifecycle of your managed objects with well-defined hooks for initialization and execution of your domain model. Additionally, since you are presumably already using Spring for your business layer, it reduces the conceptual overhead of having to maintain two different managed bean models.

In doing pure JSF development, you will quickly find that request scope is not long-lived enough for storing conversational model objects that drive complex event-driven views. In JSF the usual option is to begin putting things into session scope, with the extra burden of needing to clean the objects up before progressing to another view or functional area of the application. What is really needed is a managed scope that is somewhere between request and session scope. JSF provides flash and view scopes that can be accessed programmatically via UIViewRoot.getViewMap(). Spring Web Flow provides access to flash, view, flow, and conversation scopes. These scopes are seamlessly integrated through JSF variable resolvers and work the same in all JSF applications.

Using Flow Variables

The easiest and most natural way to declare and manage the model is through the use of flow variables. You can declare these variables at the beginning of the flow:

```
<var name="searchCriteria" class="com.mycompany.myapp.hotels.search.SearchCriteria"/>
```

and then reference this variable in one of the flow's JSF view templates through EL:

```
<h:inputText id="searchString" value="#{searchCriteria.searchString}"/>
```

Note that you do not need to prefix the variable with its scope when referencing it from the template (though you can do so if you need to be more specific). As with standard JSF beans, all available scopes will be searched for a matching variable, so you could change the scope of the variable in your flow definition without having to modify the EL expressions that reference it.

You can also define view instance variables that are scoped to the current view and get cleaned up automatically upon transitioning to another view. This is quite useful with JSF as views are often constructed to handle multiple in-page events across many requests before transitioning to another view.

To define a view instance variable, you can use the `var` element inside a `view-state` definition:

```
<view-state id="enterSearchCriteria">
 <var name="searchCriteria" class="com.mycompany.myapp.hotels.search.SearchCriteria"/>
</view-state>
```

Using Scoped Spring Beans

Though defining autowired flow instance variables provides nice modularization and readability, occasions may arise where you want to utilize the other capabilities of the Spring container such as AOP. In these cases, you can define a bean in your Spring ApplicationContext and give it a specific web flow scope:

```
<bean id="searchCriteria" class="com.mycompany.myapp.hotels.search.SearchCriteria" scope="flow"/>
```

The major difference with this approach is that the bean will not be fully initialized until it is first accessed via an EL expression. This sort of lazy instantiation via EL is quite similar to how JSF managed beans are typically allocated.

Manipulating The Model

The need to initialize the model before view rendering (such as by loading persistent entities from a database) is quite common, but JSF by itself does not provide any convenient hooks for such initialization. The flow definition language provides a natural facility for this through its Actions . Spring Web Flow provides some extra conveniences for converting the outcome of an action into a JSF-specific data structure. For example:

```
<on-render>
 <evaluate expression="bookingService.findBookings(currentUser.name)"
    result="viewScope.bookings" result-type="dataModel" />
</on-render>
```

This will take the result of the `bookingService.findBookings` method an wrap it in a custom JSF DataModel so that the list can be used in a standard JSF DataTable component:

```
<h:dataTable id="bookings" styleClass="summary" value="#{bookings}" var="booking"
   rendered="#{bookings.rowCount > 0}">
 <h:column>
  <f:facet name="header">Name</f:facet>
  #{booking.hotel.name}
 </h:column>
 <h:column>
 <f:facet name="header">Confirmation number</f:facet>
  #{booking.id}
  </h:column>
 <h:column>
  <f:facet name="header">Action</f:facet>
  <h:commandLink id="cancel" value="Cancel" action="cancelBooking" />
 </h:column>
</h:dataTable>
```

Data Model Implementations

In the example above result-type="dataModel" results in the wrapping of List<Booking> with custom `DataModel` type. The custom `DataModel` provides extra conveniences such as being serializable for storage beyond request scope as well as access to the currently selected row in EL expressions. For example, on postback from a view where the action event was fired by a component within a DataTable, you can take action on the selected row's model instance:

```
<transition on="cancelBooking">
 <evaluate expression="bookingService.cancelBooking(bookings.selectedRow)" />
</transition>
```

Spring Web Flow provides two custom DataModel types: `OneSelectionTrackingListDataModel` and `ManySelectionTrackingListDataModel`. As the names indicate they keep track of one or multiple selected rows. This is done with the help of a `SelectionTrackingActionListener` listener, which responds to JSF action events and invokes the appopriate methods on the `SelectinAware` data models to record the currently clicked row.

To understand how this is configured, keep in mind the `FacesConversionService` registers a `DataModelConverter` against the alias "dataModel" on startup. When result-type="dataModel" is used in a flow definition it causes the `DataModelConverter` to be used. The converter then wraps the given List with an instance of `OneSelectionTrackingListDataModel`. To use the `ManySelectionTrackingListDataModel` you will need to register your own custom converter.

13.5. Handling JSF Events With Spring Web Flow

Spring Web Flow allows you to handle JSF action events in a decoupled way, requiring no direct dependencies in your Java code on JSF API's. In fact, these events can often be handled completely in the flow definiton language without requiring any custom Java action code at all. This allows for a more agile development process since the artifacts being manipulated in wiring up events (JSF view templates and SWF flow definitions) are instantly refreshable without requiring a build and re-deploy of the whole application.

Handling JSF In-page Action Events

A simple but common case in JSF is the need to signal an event that causes manipulation of the model in some way and then redisplays the same view to reflect the changed state of the model. The flow definition language has special support for this in the `transition` element.

A good example of this is a table of paged list results. Suppose you want to be able to load and display only a portion of a large result list, and allow the user to page through the results. The initial `view-state` definition to load and display the list would be:

```
<view-state id="reviewHotels">
 <on-render>
  <evaluate expression="bookingService.findHotels(searchCriteria)"
     result="viewScope.hotels" result-type="dataModel" />
 </on-render>
</view-state>
```

You construct a JSF DataTable that displays the current `hotels` list, and then place a "More Results" link below the table:

```
<h:commandLink id="nextPageLink" value="More Results" action="next"/>
```

This commandLink signals a "next" event from its action attribute. You can then handle the event by adding to the `view-state` definition:

```
<view-state id="reviewHotels">
 <on-render>
  <evaluate expression="bookingService.findHotels(searchCriteria)"
   result="viewScope.hotels" result-type="dataModel" />
 </on-render>
 <transition on="next">
  <evaluate expression="searchCriteria.nextPage()" />
 </transition>
</view-state>
```

Here you handle the "next" event by incrementing the page count on the searchCriteria instance. The `on-render` action is then called again with the updated criteria, which causes the next page of results to be loaded into the DataModel. The same view is re-rendered since there was no `to` attribute on the `transition` element, and the changes in the model are reflected in the view.

Handling JSF Action Events

The next logical level beyond in-page events are events that require navigation to another view, with some manipulation of the model along the way. Achieving this with pure JSF would require adding a navigation rule to faces-config.xml and likely some intermediary Java code in a JSF managed bean (both tasks requiring a re-deploy). With the flow defintion language, you can handle such a case concisely in one place in a quite similar way to how in-page events are handled.

Continuing on with our use case of manipulating a paged list of results, suppose we want each row in the displayed DataTable to contain a link to a detail page for that row instance. You can add a column to the table containing the following `commandLink` component:

```
<h:commandLink id="viewHotelLink" value="View Hotel" action="select"/>
```

This raises the "select" event which you can then handle by adding another `transition` element to the existing `view-state` :

```
<view-state id="reviewHotels">
 <on-render>
  <evaluate expression="bookingService.findHotels(searchCriteria)"
   result="viewScope.hotels" result-type="dataModel" />
 </on-render>
 <transition on="next">
  <evaluate expression="searchCriteria.nextPage()" />
 </transition>
 <transition on="select" to="reviewHotel">
   <set name="flowScope.hotel" value="hotels.selectedRow" />
 </transition>
</view-state>
```

Here the "select" event is handled by pushing the currently selected hotel instance from the DataTable into flow scope, so that it may be referenced by the "reviewHotel" `view-state` .

Performing Model Validation

JSF provides useful facilities for validating input at field-level before changes are applied to the model, but when you need to then perform more complex validation at the model-level after the updates have been applied, you are generally left with having to add more custom code to your JSF action methods in the managed bean. Validation of this sort is something that is generally a responsibility of the domain model itself, but it is difficult to get any error messages propagated back to the view without introducing an undesirable dependency on the JSF API in your domain layer.

With Web Flow, you can utilize the generic and low-level `MessageContext` in your business code and any messages added there will then be available to the `FacesContext` at render time.

For example, suppose you have a view where the user enters the necessary details to complete a hotel booking, and you need to ensure the Check In and Check Out dates adhere to a given set of business rules. You can invoke such model-level validation from a `transition` element:

```
<view-state id="enterBookingDetails">
 <transition on="proceed" to="reviewBooking">
  <evaluate expression="booking.validateEnterBookingDetails(messageContext)" />
 </transition>
</view-state>
```

Here the "proceed" event is handled by invoking a model-level validation method on the booking instance, passing the generic `MessageContext` instance so that messages may be recorded. The messages can then be displayed along with any other JSF messages with the `h:messages` component,

Handling Ajax Events In JSF

JSF provides built-in support for sending Ajax requests and performing partial processing and rendering on the server-side. You can specify a list of id's for partial rendering through the <f:ajax> facelets tag.

In Spring Web Flow you also have the option to specify the ids to use for partial rendering on the server side with the render action:

```
<view-state id="reviewHotels">
 <on-render>
  <evaluate expression="bookingService.findHotels(searchCriteria)"
     result="viewScope.hotels" result-type="dataModel" />
 </on-render>
 <transition on="next">
  <evaluate expression="searchCriteria.nextPage()" />
  <render fragments="hotels:searchResultsFragment" />
 </transition>
</view-state>
```

13.6. Embedding a Flow On a Page

By default when a flow enters a view state, it executes a client-side redirect before rendering the view. This approach is known as POST-REDIRECT-GET. It has the advantage of separating the form processing for one view from the rendering of the next view. As a result the browser Back and Refresh buttons work seamlessly without causing any browser warnings.

Normally the client-side redirect is transparent from a user's perspective. However, there are situations where POST-REDIRECT-GET may not bring the same benefits. For example sometimes it may be useful to embed a flow on a page and drive it via Ajax requests refreshing only the area of the page where the flow is rendered. Not only is it unnecessary to use client-side redirects in this case, it is also not the desired behavior with regards to keeping the surrounding content of the page intact.

To indicate a flow should execute in "page embedded" mode all you need to do is pass an extra flow input attribute called "mode" with a value of "embedded". Below is an example of a top-level container flow invoking a sub-flow in an embedded mode:

```
<subflow-state id="bookHotel" subflow="booking">
 <input name="mode" value="'embedded'"/>
</subflow-state>
```

When launched in "page embedded" mode the sub-flow will not issue flow execution redirects during Ajax requests.

If you'd like to see examples of an embedded flow please refer to the webflow-primefaces-showcase project. You can check out the source code locally, build it as you would a Maven project, and import it into Eclipse:

```
cd some-directory
svn co https://src.springframework.org/svn/spring-samples/webflow-primefaces-showcase
cd webflow-primefaces-showcase
mvn package
# import into Eclipse
```

The specific example you need to look at is under the "Advanced Ajax" tab and is called "Top Flow with Embedded Sub-Flow".

13.7. Redirect In Same State

By default Web Flow does a client-side redirect even it it remains in the same view state as long as the current request is not an Ajax request. This is quite useful after form validation failures for example. If the user hits Refresh or Back they won't see any browser warnings. They would if the Web Flow didn't do a redirect.

This can lead to a problem specific to JSF environments where a specific Sun Mojarra listener component caches the FacesContext assuming the same instance is available throughout the JSF lifecycle. In Web Flow however the render phase is temporarily put on hold and a client-side redirect executed.

The default behavior of Web Flow is desirable and it is unlikely JSF applications will experience the issue. This is because Ajax is often enabled the default in JSF component libraries and Web Flow does not redirect during Ajax requests. However if you experience this issue you can disable client-side redirects within the same view as follows:

```
<webflow:flow-executor id="flowExecutor">
 <webflow:flow-execution-attributes>
  <webflow:redirect-in-same-state value="false"/>
 </webflow:flow-execution-attributes>
</webflow:flow-executor>
```

13.8. Handling File Uploads with JSF

Most JSF component providers include some form of 'file upload' component. Generally when working with these components JSF must take complete control of parsing multi-part requests and Spring MVC's `MultipartResolver` cannot be used.

Spring Web Flow has been tested with file upload components from PrimeFaces. Check the documentation of your JSF component library for other providers to see how to configure file upload.

File Uploads with PrimeFaces

PrimeFaces provides a `<p:fileUpload>` component for uploading files. In order to use the component you need to configure the `org.primefaces.webapp.filter.FileUploadFilter` servlet filter. The filter needs to be configured against Spring MVC's `DispatcherServlet` in your `web.xml`:

```
<filter>
 <filter-name>PrimeFaces FileUpload Filter</filter-name>
 <filter-class>org.primefaces.webapp.filter.FileUploadFilter</filter-class>
</filter>
<filter-mapping>
 <filter-name>PrimeFaces FileUpload Filter</filter-name>
 <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
</filter-mapping>

<context-param>
 <param-name>primefaces.UPLOADER</param-name>
 <param-value>commons</param-value>
</context-param>
```

For more details refer to the PrimeFaces documentation.

13.9. Using the Spring Security Facelets Tag Library

To use the library you'll need to create a `.taglib.xml` file and register it in `web.xml`.

Create the file `/WEB-INF/springsecurity.taglib.xml` with the following content:

```
<?xml version="1.0"?>
<!DOCTYPE facelet-taglib PUBLIC
"-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
"http://java.sun.com/dtd/facelet-taglib_1_0.dtd">
<facelet-taglib>
 <namespace>http://www.springframework.org/security/tags</namespace>
 <tag>
  <tag-name>authorize</tag-name>
  <handler-class>org.springframework.faces.security.FaceletsAuthorizeTagHandler</handler-class>
 </tag>
 <function>
  <function-name>areAllGranted</function-name>
  <function-class>org.springframework.faces.security.FaceletsAuthorizeTagUtils</function-class>
  <function-signature>boolean areAllGranted(java.lang.String)</function-signature>
 </function>
 <function>
  <function-name>areAnyGranted</function-name>
  <function-class>org.springframework.faces.security.FaceletsAuthorizeTagUtils</function-class>
  <function-signature>boolean areAnyGranted(java.lang.String)</function-signature>
 </function>
 <function>
  <function-name>areNotGranted</function-name>
  <function-class>org.springframework.faces.security.FaceletsAuthorizeTagUtils</function-class>
  <function-signature>boolean areNotGranted(java.lang.String)</function-signature>
 </function>
 <function>
  <function-name>isAllowed</function-name>
  <function-class>org.springframework.faces.security.FaceletsAuthorizeTagUtils</function-class>
  <function-signature>boolean isAllowed(java.lang.String, java.lang.String)</function-signature>
 </function>
</facelet-taglib>
```

Next, register the above file taglib in web.xml:

```
<context-param>
 <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
 <param-value>/WEB-INF/springsecurity.taglib.xml</param-value>
</context-param>
```

Now you are ready to use the tag library in your views. You can use the authorize tag to include nested content conditionally:

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ui="http://java.sun.com/jsf/facelets"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:sec="http://www.springframework.org/security/tags">

 <sec:authorize ifAllGranted="ROLE_FOO, ROLE_BAR">
  Lorem ipsum dolor sit amet
 </sec:authorize>

 <sec:authorize ifNotGranted="ROLE_FOO, ROLE_BAR">
  Lorem ipsum dolor sit amet
 </sec:authorize>

 <sec:authorize ifAnyGranted="ROLE_FOO, ROLE_BAR">
  Lorem ipsum dolor sit amet
 </sec:authorize>

</ui:composition>
```

You can also use one of several EL functions in the rendered or other attribute of any JSF component:

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ui="http://java.sun.com/jsf/facelets"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:sec="http://www.springframework.org/security/tags">

 <!-- Rendered only if user has all of the listed roles -->
 <h:outputText value="Lorem ipsum dolor sit amet" rendered="#{sec:areAllGranted('ROLE_FOO, ROLE_BAR')}"/
>

 <!-- Rendered only if user does not have any of the listed roles -->
 <h:outputText value="Lorem ipsum dolor sit amet" rendered="#{sec:areNotGranted('ROLE_FOO, ROLE_BAR')}"/
>

 <!-- Rendered only if user has any of the listed roles -->
 <h:outputText value="Lorem ipsum dolor sit amet" rendered="#{sec:areAnyGranted('ROLE_FOO, ROLE_BAR')}"/
>

 <!-- Rendered only if user has access to given HTTP method/URL as defined in Spring Security
 configuration -->
 <h:outputText value="Lorem ipsum dolor sit amet" rendered="#{sec:isAllowed('/secured/foo', 'POST')}"/>

</ui:composition>
```

13.10. Third-Party Component Library Integration

The Spring Web Flow JSF integration strives to be compatible with any third-party JSF component library. By honoring all of the standard semantics of the JSF specification within the SWF-driven JSF lifecycle, third-party libraries in general should "just work". The main thing to remember is that configuration in web.xml will change slightly since Web Flow requests are not routed through the standard FacesServlet. Typically, anything that is traditionally mapped to the FacesServlet should be mapped to the Spring DispatcherServlet instead. (You can also map to both if for example you are migrating a legacy JSF application page-by-page.).

14. Testing flows

14.1. Introduction

This chapter shows you how to test flows.

14.2. Extending AbstractXmlFlowExecutionTests

To test the execution of a XML-based flow definition, extend `AbstractXmlFlowExecutionTests`:

```
public class BookingFlowExecutionTests extends AbstractXmlFlowExecutionTests {

}
```

14.3. Specifying the path to the flow to test

At a minimum, you must override `getResource(FlowDefinitionResourceFactory)` to return the path to the flow you wish to test:

```
@Override
protected FlowDefinitionResource getResource(FlowDefinitionResourceFactory resourceFactory) {
 return resourceFactory.createFileResource("src/main/webapp/WEB-INF/hotels/booking/booking.xml");
}
```

14.4. Registering flow dependencies

If your flow has dependencies on externally managed services, also override `configureFlowBuilderContext(MockFlowBuilderContext)` to register stubs or mocks of those services:

```
@Override
protected void configureFlowBuilderContext(MockFlowBuilderContext builderContext) {
 builderContext.registerBean("bookingService", new StubBookingService());
}
```

If your flow extends from another flow, or has states that extend other states, also override `getModelResources(FlowDefinitionResourceFactory)` to return the path to the parent flows.

```
@Override
protected FlowDefinitionResource[] getModelResources(FlowDefinitionResourceFactory resourceFactory) {
return new FlowDefinitionResource[] {
    resourceFactory.createFileResource("src/main/webapp/WEB-INF/common/common.xml")
};
}
```

14.5. Testing flow startup

Have your first test exercise the startup of your flow:

```
public void testStartBookingFlow() {

 Booking booking = createTestBooking();

 MutableAttributeMap input = new LocalAttributeMap();
 input.put("hotelId", "1");
 MockExternalContext context = new MockExternalContext();
 context.setCurrentUser("keith");
 startFlow(input, context);

 assertCurrentStateEquals("enterBookingDetails");
 assertTrue(getRequiredFlowAttribute("booking") instanceof Booking);
}
```

Assertions generally verify the flow is in the correct state you expect.

14.6. Testing flow event handling

Define additional tests to exercise flow event handling behavior. You goal should be to exercise all paths through the flow. You can use the convenient `setCurrentState(String)` method to jump to the flow state where you wish to begin your test.

```
public void testEnterBookingDetails_Proceed() {

 setCurrentState("enterBookingDetails");

 getFlowScope().put("booking", createTestBooking());

 MockExternalContext context = new MockExternalContext();
 context.setEventId("proceed");
 resumeFlow(context);

 assertCurrentStateEquals("reviewBooking");
}
```

14.7. Mocking a subflow

To test calling a subflow, register a mock implementation of the subflow that asserts input was passed in correctly and returns the correct outcome for your test scenario.

```
public void testBookHotel() {

 setCurrentState("reviewHotel");

 Hotel hotel = new Hotel();
 hotel.setId(1L);
 hotel.setName("Jameson Inn");
 getFlowScope().put("hotel", hotel);

 getFlowDefinitionRegistry().registerFlowDefinition(createMockBookingSubflow());

 MockExternalContext context = new MockExternalContext();
 context.setEventId("book");
 resumeFlow(context);

 // verify flow ends on 'bookingConfirmed'
 assertFlowExecutionEnded();
 assertFlowExecutionOutcomeEquals("finish");
}

public Flow createMockBookingSubflow() {
 Flow mockBookingFlow = new Flow("booking");
 mockBookingFlow.setInputMapper(new Mapper() {
  public MappingResults map(Object source, Object target) {
   // assert that 1L was passed in as input
   assertEquals(1L, ((AttributeMap) source).get("hotelId"));
   return null;
  }
 });
 // immediately return the bookingConfirmed outcome so the caller can respond
 new EndState(mockBookingFlow, "bookingConfirmed");
 return mockBookingFlow;
}
```

Appendix A. Flow Definition Language 1.0 to 2.0 Mappings

The flow definition language has changed since the 1.0 release. This is a listing of the language elements in the 1.0 release, and how they map to elements in the 2.0 release. While most of the changes are semantic, there are a few structural changes. Please see the upgrade guide for more details about changes between Web Flow 1.0 and 2.0.

Table A.1. Mappings

SWF 1.0		**SWF 2.0**		**Comments**
action		*		use <evaluate />
	bean		*	
	name		*	
	method		*	
action-state		*action-state*		
	id		id	
	*		parent	
argument		*		use <evaluate expression="func(arg1, arg2, ...)"/>
	expression			
	parameter-type			
attribute		*attribute*		
	name		name	
	type		type	
	value		value	
attribute-mapper		*		input and output elements can be in flows or subflows directly
	bean		*	now subflow-attribute-mapper attribute on subflow-state
bean-action		*		use <evaluate />
	bean		*	
	name		*	
	method		*	
decision-state		*decision-state*		

SWF 1.0		**SWF 2.0**		**Comments**
	id		id	
	*		parent	
end-actions		*on-end*		
end-state		*end-state*		
	id		id	
	view		view	
	*		parent	
	*		commit	
entry-actions		*on-entry*		
evaluate-action		*evaluate*		
	expression		expression	
	name		*	use <evaluate ...> <attribute name="name" value="..." /> </evaluate>
	*		result	
	*		result-type	
evaluation-result		*		use <evaluate result="..." />
	name		*	
	scope		*	
exception-handler		*exception-handler*		
	bean		bean	
exit-actions		*on-exit*		
flow		*flow*		
	*		start-state	
	*		parent	
	*		abstract	
global-transitions		*global-transitions*		
if		*if*		
	test		test	
	then		then	
	else		else	
import		*bean-import*		

SWF 1.0		**SWF 2.0**		**Comments**
	resource		resource	
inline-flow		*		convert to new top-level flow
	id		*	
input-attribute		*input*		
	name		name	
	scope		*	prefix name with scope <input name="flowScope.foo" />
	required		required	
	*		type	
	*		value	
input-mapper		*		inputs can be in flows and subflows directly
mapping		*input or output*		
	source		name or value	name when in flow element, value when in subflow-state element
	target		name or value	value when in flow element, name when in subflow-state element
	target-collection		*	no longer supported
	from		*	detected automatically
	to		type	
	required		required	
method-argument		*		use <evaluate expression="func(arg1, arg2, ...)"/>
method-result		*		use <evaluate result="..." />
	name		*	
	scope		*	
output-attribute		*output*		
	name		name	
	scope		*	prefix name with scope <output name="flowScope.foo" />
	required		required	
	*		type	
	*		value	

SWF 1.0		**SWF 2.0**		**Comments**
output-mapper		*		output can be in flows and subflows directly
render-actions		*on-render*		
set		*set*		
	attribute		name	
	scope		*	prefix name with scope <set name="flowScope.foo" />
	value		value	
	name		*	use <set ...> <attribute name=”name” value="..." /> </set>
	*		type	
start-actions		*on-start*		
start-state		*		now <flow start-state="...">, or defaults to the first state in the flow
	idref		*	
subflow-state		*subflow-state*		
	id		id	
	flow		subflow	
	*		parent	
	*		subflow-attribute-mapper	
transition		*transition*		
	on		on	
	on-exception		on-exception	
	to		to	
	*		bind	
	*		validate	
	*		history	
value		*value*		
var		*var*		
	name		name	
	class		class	
	scope		*	always flow scope

SWF 1.0		**SWF 2.0**		**Comments**
	bean		*	all Spring beans can be resolved with EL
view-state		*view-state*		
	id		id	
	view		view	
	*		parent	
	*		redirect	
	*		popup	
	*		model	
	*		history	
*		*persistence-context*		
*		*render*		
	*		fragments	
*		*secured*		
	*		attributes	
	*		match	