

Spring Web Flow Reference Guide

**Keith Donald
Erwin Vervaet
Jeremy Grelle
Scott Andrews
Rossen Stoyanchev**

Version 2.0.9

Published

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Table of Contents

Preface	vii
1. Introduction	1
1.1. What this guide covers	1
1.2. What Web Flow requires to run	1
1.3. Where to get support	1
1.4. Where to follow development	1
1.5. How to access Web Flow artifacts from Maven Central	1
1.6. How to access Web Flow artifacts from the SpringSource Bundle Repository	2
1.7. How to access nightly builds	4
2. Defining Flows	7
2.1. Introduction	7
2.2. What is a flow?	7
2.3. What is the makeup of a typical flow?	8
2.4. How are flows authored?	8
2.5. Essential language elements	9
2.6. Actions	10
2.7. Input/Output Mapping	12
2.8. Variables	14
2.9. Calling subflows	14
3. Expression Language (EL)	17
3.1. Introduction	17
3.2. Supported EL implementations	17
3.3. EL portability	17
3.4. EL usage	17
3.5. Special EL variables	19
3.6. Scope searching algorithm	21
4. Rendering views	23
4.1. Introduction	23
4.2. Defining view states	23
4.3. Specifying view identifiers	23
4.4. View scope	24
4.5. Executing render actions	25
4.6. Binding to a model	25
4.7. Performing type conversion	26
4.8. Suppressing binding	27
4.9. Specifying bindings explicitly	27
4.10. Validating a model	28
4.11. Suppressing validation	30
4.12. Executing view transitions	31
4.13. Working with messages	32
4.14. Displaying popups	34
4.15. View backtracking	34
5. Executing actions	35
5.1. Introduction	35

5.2. Defining action states	35
5.3. Defining decision states	35
5.4. Action outcome event mappings	36
5.5. Action implementations	36
5.6. Action exceptions	37
5.7. Other Action execution examples	38
6. Flow Managed Persistence	43
6.1. Introduction	43
6.2. FlowScoped PersistenceContext	43
7. Securing Flows	45
7.1. Introduction	45
7.2. How do I secure a flow?	45
7.3. The secured element	45
7.4. The SecurityFlowExecutionListener	46
7.5. Configuring Spring Security	47
8. Flow Inheritance	49
8.1. Introduction	49
8.2. Is flow inheritance like Java inheritance?	49
8.3. Types of Flow Inheritance	49
8.4. Abstract flows	50
8.5. Inheritance Algorithm	50
9. System Setup	53
9.1. Introduction	53
9.2. webflow-config.xsd	53
9.3. Basic system configuration	53
9.4. flow-registry options	54
9.5. flow-executor options	57
10. Spring MVC Integration	59
10.1. Introduction	59
10.2. Configuring web.xml	59
10.3. Dispatching to flows	59
10.4. Implementing custom FlowHandlers	60
10.5. View Resolution	63
10.6. Signaling an event from a View	63
11. Spring JavaScript Quick Reference	65
11.1. Introduction	65
11.2. Serving Javascript Resources	65
11.3. Including Spring Javascript in a Page	65
11.4. Spring Javascript Decorations	66
11.5. Handling Ajax Requests	67
12. JSF Integration	71
12.1. Introduction	71
12.2. Spring-centric Integration Approach	71
12.3. Configuring web.xml	72
12.4. Configuring Web Flow to render JSF views	73
12.5. Configuring faces-config.xml	74
12.6. Replacing the JSF Managed Bean Facility	74
12.7. Handling JSF Events With Spring Web Flow	76
12.8. Enhancing The User Experience With Rich Web Forms	80

12.9. Third-Party Component Library Integration	81
13. Portlet Integration	85
13.1. Introduction	85
13.2. Configuring web.xml and portlet.xml	85
13.3. Configuring Spring	85
13.4. Portlet Views	87
13.5. Portlet Modes and Window States	87
13.6. Issues in a Portlet Environment	88
14. Testing flows	89
14.1. Introduction	89
14.2. Extending AbstractXmlFlowExecutionTests	89
14.3. Specifying the path to the flow to test	89
14.4. Registering flow dependencies	89
14.5. Testing flow startup	90
14.6. Testing flow event handling	90
14.7. Mocking a subflow	90
15. Upgrading from 1.0	93
15.1. Introduction	93
15.2. Flow Definition Language	93
15.3. Web Flow Configuration	94
15.4. New Web Flow Concepts	96
A. Flow Definition Language 1.0 to 2.0 Mappings	99

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 1.4 or higher

Spring 2.5.6 or higher

1.3. Where to get support

Professional from-the-source support on Spring Web Flow is available from [SpringSource](#), the company behind Spring, and [Ervacon](#), operated by Web Flow project co-founder Erwin Vervaeke.

1.4. Where to follow development

You can help make Web Flow best serve the needs of the Spring community by interacting with developers at the [Spring Community Forums](#).

Report bugs and influence the Web Flow project roadmap using the [Spring Issue Tracker](#).

Subscribe to the [Spring Community Portal](#) for the latest Spring news and announcements.

Visit the [Web Flow Project Home](#) for more resources on the project.

1.5. 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 dependencies in your pom:

```
<code><!-- Empty box for Maven dependencies --></code>
```

```
<dependency>
  <groupId>org.springframework.webflow</groupId>
  <artifactId>spring-binding</artifactId>
  <version>2.0.9.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.webflow</groupId>
  <artifactId>spring-js</artifactId>
  <version>2.0.9.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.webflow</groupId>
  <artifactId>spring-webflow</artifactId>
  <version>2.0.9.RELEASE</version>
</dependency>
```

And if using JavaServerFaces:

```
<dependency>
  <groupId>org.springframework.webflow</groupId>
  <artifactId>spring-faces</artifactId>
  <version>2.0.9.RELEASE</version>
</dependency>
```

1.6. How to access Web Flow artifacts from the SpringSource Bundle Repository

Each jar in the Web Flow distribution is also available in the [SpringSource Enterprise Bundle Repository](#). Use this repository when you wish to run Spring Web Flow in an OSGi environment such as the [SpringSource dm Server](#). All jars obtained from the SpringSource Bundle Repository are OSGi-ready.

Accessing Web Flow bundles with Maven

To access bundles using Maven, add the following repositories to your Maven pom:

```
<repository>
  <id>com.springsource.repository.bundles.release</id>
  <name>SpringSource Enterprise Bundle Repository - SpringSource Releases</name>
  <url>http://repository.springsource.com/maven/bundles/release</url>
</repository>

<repository>
  <id>com.springsource.repository.bundles.external</id>
  <name>SpringSource Enterprise Bundle Repository - External Releases</name>
  <url>http://repository.springsource.com/maven/bundles/external</url>
</repository>
```

Then declare the following dependencies:

```
<dependency>
  <groupId>org.springframework.webflow</groupId>
  <artifactId>org.springframework.binding</artifactId>
  <version>2.0.9.RELEASE</version>
</dependency>

<dependency>
```

```

<groupId>org.springframework.webflow</groupId>
<artifactId>org.springframework.js</artifactId>
<version>2.0.9.RELEASE</version>
</dependency>

<dependency>
<groupId>org.springframework.webflow</groupId>
<artifactId>org.springframework.webflow</artifactId>
<version>2.0.9.RELEASE</version>
</dependency>

```

And if using JavaServerFaces:

```

<dependency>
<groupId>org.springframework.webflow</groupId>
<artifactId>org.springframework.faces</artifactId>
<version>2.0.9.RELEASE</version>
</dependency>

```

Note the Web Flow artifacts in the SpringSource Bundle Repository are indexed under different ids because their transitive dependencies are different than the Maven Central artifacts. The difference is the transitive jars such as commons-logging have been patched by SpringSource to add the metadata required to make them OSGi-compatible.

Accessing Web Flow bundles with Ivy

To access bundles using Ivy, add the following repositories to your Ivy config:

```

<url name="com.springsource.repository.bundles.release">
  <ivy pattern="http://repository.springsource.com/ivy/bundles/release/
    [organisation]/[module]/[revision]/[artifact]-[revision].[ext]" />
  <artifact pattern="http://repository.springsource.com/ivy/bundles/release/
    [organisation]/[module]/[revision]/[artifact]-[revision].[ext]" />
</url>

<url name="com.springsource.repository.bundles.external">
  <ivy pattern="http://repository.springsource.com/ivy/bundles/external/
    [organisation]/[module]/[revision]/[artifact]-[revision].[ext]" />
  <artifact pattern="http://repository.springsource.com/ivy/bundles/external/
    [organisation]/[module]/[revision]/[artifact]-[revision].[ext]" />
</url>

```

Then declare the following dependencies:

```

<dependency org="org.springframework.webflow" name="org.springframework.binding"
  rev="2.0.9.RELEASE" conf="compile->runtime" />
<dependency org="org.springframework.webflow" name="org.springframework.js"
  rev="2.0.9.RELEASE" conf="compile->runtime" />
<dependency org="org.springframework.webflow" name="org.springframework.webflow"
  rev="2.0.9.RELEASE" conf="compile->runtime" />

```

And if using JavaServerFaces:

```

<dependency org="org.springframework.webflow" name="org.springframework.faces"
  rev="2.0.9.RELEASE" conf="compile->runtime" />

```

Accessing the dm Server Web Flow library

A dm Server library for Web Flow is also available if you are deploying to a dm Server environment. Import this library in your MANIFEST.mf to automatically import all Web Flow bundles. To access the library, add the following repository:

```
<repository>
  <id>com.springsource.repository.libraries.release</id>
  <name>SpringSource Enterprise Bundle Repository - SpringSource Library Releases</name>
  <url>http://repository.springsource.com/maven/libraries/release</url>
</repository>
```

And declare the following dependency:

```
<dependency>
  <groupId>org.springframework.webflow</groupId>
  <artifactId>org.springframework.webflow-library</artifactId>
  <type>libd</type>
  <version>2.0.9.RELEASE</version>
</dependency>
```

1.7. How to access nightly builds

Nightly snapshots of Web Flow development branches are available using Maven, and distribution zips are also available for download. 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.

If using Maven, you may obtain snapshots from either the SpringSource-managed Maven Central-compatible repository or the SpringSource Enterprise Bundle Repository. Use the [Maven Central-compatible snapshot repository](#) when your project obtains its other open source dependencies from Maven Central. Use the [Spring Source Enterprise Bundle Snapshot Repository](#) when you wish to run Web Flow in an OSGi environment.

Accessing snapshots from the Maven-central compatible repository

Add the following repository your pom:

```
<repository>
  <id>org.springsource.maven.snapshot</id>
  <name>SpringSource Maven Central-compatible Snapshot Repository</name>
  <url>http://maven.springframework.org/snapshot</url>
</repository>
```

Then declare the following dependencies:

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

<dependency>
  <groupId>org.springframework.webflow</groupId>
```

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

And if using JavaServerFaces:

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

Accessing snapshots from the SpringSource Enterprise Bundle Repository

Add the following repository your pom:

```
<repository>
<id>com.springsource.repository.bundles.snapshot</id>
<name>SpringSource Enterprise Bundle Snapshot Repository</name>
<url>http://repository.springsource.com/maven/bundles/snapshot</url>
</repository>
```

Then declare the following dependencies:

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

And if using JavaServerFaces:

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

Accessing snapshot distribution archives

Get the snapshot zip with the most recent CI build number from the [Web Flow snapshot](#)

[download area.](#)

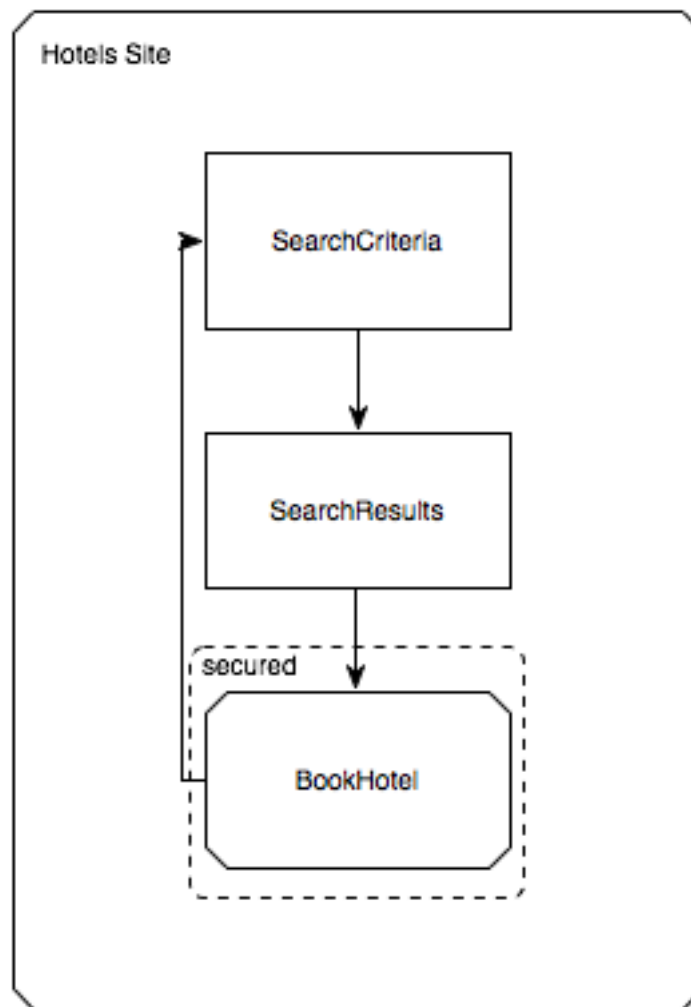
2. Defining Flows

2.1. 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.

2.2. 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:

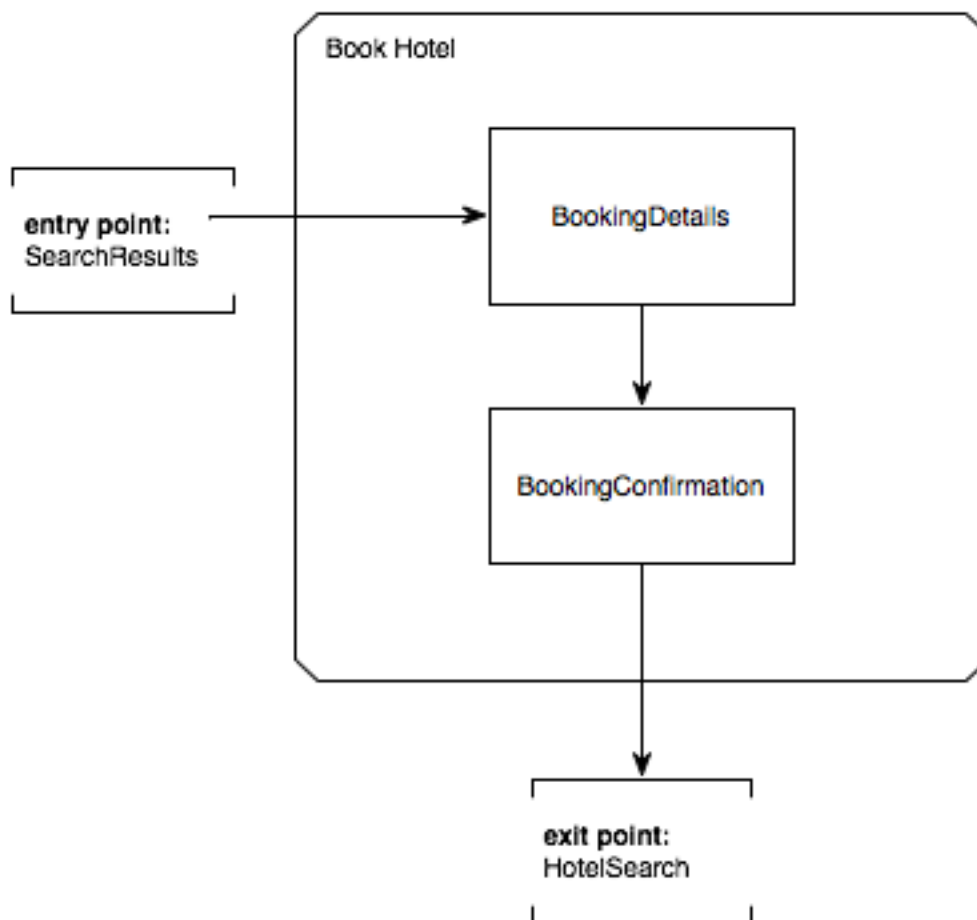


Site Map illustrating a reference to a flow

2.3. 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

2.4. 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.

2.5. 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-2.0.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-2.0.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>
```

2.6. 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-2.0.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.

2.7. 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 coercion 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-2.0.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.

2.8. 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.

2.9. 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

Simply refer to a subflow output attribute by its name within a outcome transition:

```
<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-2.0.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.

3. Expression Language (EL)

3.1. Introduction

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

3.2. Supported EL implementations

Unified EL

Web Flow attempts to use the [Unified EL](#) by default. `jboss-el` is currently the default EL implementation. When found in your classpath along with the `el-api`, it will be used automatically. You can find the JBoss EL jar in the [SpringSource Bundle Repository](#).



Note

The `el-api` dependency is typically *provided* by your web container. Tomcat 6 includes it, for example.

OGNL

[OGNL](#) is the other EL supported by Web Flow 2. OGNL is the EL most familiar to Web Flow version 1.0 users. To use ognl, simply include `ognl` in your classpath instead of `jboss-el`. Please refer to the [OGNL language guide](#) for specifics on its EL syntax.

3.3. EL portability

In general, you will find the Unified EL and OGNL have a very similar syntax. For basic variable resolution, property access, and method invocation the syntax is identical. We recommend adhering to Unified EL syntax whenever possible, and only relying on proprietary EL features when needed.

3.4. EL usage

EL is used for many things within a flow, including:

1. Accessing data provided by the client, such as flow input attributes and request parameters.

2. Accessing internal data structures such as `flowScope`.
3. Invoking methods on Spring beans.
4. Resolving constructs such as state transition criteria, subflow ids, and view names.

Views rendered by flows typically access flow data structures using EL as well.

Expression types

There are basically two types of expressions in Web Flow.

Standard eval expressions

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

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

The expression above is a standard expression that invokes the `nextPage` method on the `searchCriteria` variable when evaluated. Attempting to enclose this expression in special eval delimiters like `${ }` or `#{ }` will result in an `IllegalArgumentException`.



Note

We view use of special eval delimiters as redundant in this context, as the only acceptable value for the `expression` attribute is a single eval expression string.

Template expressions

The second type of expression is a "template" expression. Such expressions allow a mixing of literal text with one or more eval blocks. Each eval block is explicitly delimited 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 the literal text `error-` with the result of evaluating `externalContext.locale`. As you can see, explicit delimiters are necessary here to demarcate eval blocks within the template.

See the Web Flow XML schema for a complete listing of the XML attributes that accept standard expressions and template expressions.

3.5. Special EL variables

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

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" />
```

3.6. Scope searching algorithm

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)" />
```

If no scope is specified, like in the use of `booking` above, a scope searching algorithm will be employed. 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.

4. Rendering views

4.1. Introduction

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

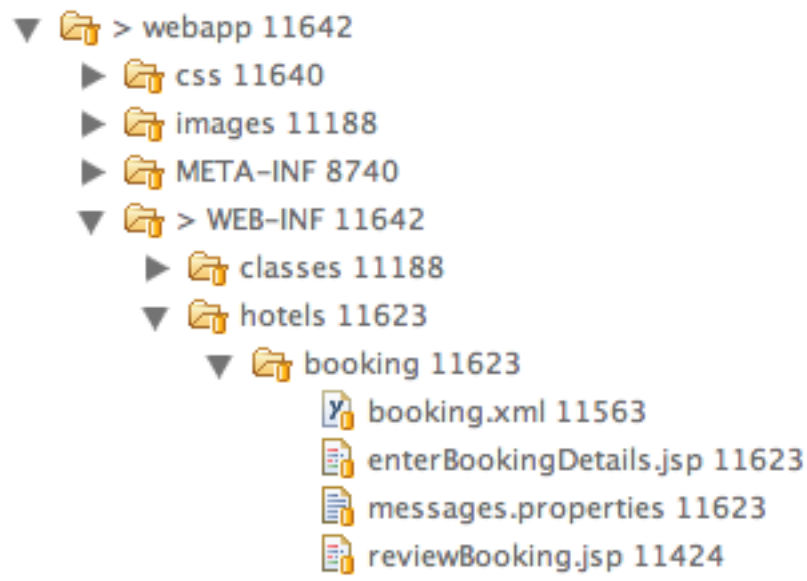
4.2. Defining view states

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

4.3. 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 an 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.

4.4. 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>
```

4.5. 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>
```

4.6. 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.

4.7. Performing type conversion

When a model binding occurs during view postback, the binding system will attempt to convert the input value to the type of the target model property if necessary. Default Converters are registered for common types such as Numbers, primitives, enums, and Dates and are applied automatically. Users also have the ability to register their own converters for user-defined types, and to override the default Converters.

Implementing a Converter

To implement your own Converter, implement the `org.springframework.binding.convert.converters.TwoWayConverter` interface. A convenient `StringToObject` base class has been provided to simplify the implementation of this interface for converters that convert from a user input `String` to a user-defined `Object` and back. Simply extend from this class and override these two methods:

```
protected abstract Object toObject(String string, Class targetClass) throws Exception;
protected abstract String toString(Object object) throws Exception;
```

`toObject(String, Class)` should convert from the input string to your object's type, and `toString(Object)` should do the reverse.

The following example shows a Converter that converts from `String` to a `MonetaryAmount` for working with currency values:

```
public class StringToMonetaryAmount extends StringToObject {
    public StringToMonetaryAmount() {
        super(MonetaryAmount.class);
    }

    @Override
    protected Object toObject(String string, Class targetClass) {
        return MonetaryAmount.valueOf(string);
    }

    @Override
    protected String toString(Object object) {
```

```
    MonetaryAmount amount = (MonetaryAmount) object;  
    return amount.toString();  
  }  
}
```

Review the pre-built converters in the `org.springframework.binding.convert.converters` package to see more examples of Converter implementations.

Registering a Converter

To install your own Converter or override any of the default Converters, extend from `org.springframework.binding.convert.service.DefaultConversionService` and override the `addDefaultConverters()` method. Use the `addConverter(Converter)` method to register the primary Converter to use to convert between two types, such as a `String` and a `MonetaryAmount`. Optionally use the `addConverter(String, Converter)` method to register alternate converters for the same type pair; for example, to support formatting a `java.util.Date` as a `String` in several different ways.

Each alternate Converter is indexed by a unique `converterId` that can be referenced when configuring a model binding. When no converter id is referenced explicitly by a binding, the primary Converter between the two types is always used.

The `ConversionService` is the object Web Flow consults at runtime to lookup conversion executors to convert from one type to another. There is generally one `ConversionService` per application. See the [System Setup](#) section for documentation on how to configure an extended `ConversionService` implementation that registers custom Converters to apply application-wide. Also consult the [Convert API](#) documentation for more information.

4.8. 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>
```

4.9. Specifying bindings explicitly

Use the `binder` element to configure the exact set of model bindings usable by the view. This is particularly useful in a Spring MVC environment for restricting the set of "allowed fields" per view.

```
<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>

```

If the binder element is not specified, all public properties of the model are eligible for binding by the view. With the binder element specified, only the explicitly configured bindings are allowed.

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.

4.10. Validating a model

Model validation is driven by constraints specified against a model object. Web Flow supports enforcing such constraints programmatically.

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").
                defaultMessage("Check in date must be a future date").build());
        } else if (!checkinDate.before(checkoutDate)) {
            messages.addMessage(new MessageBuilder().error().source("checkoutDate").
                defaultMessage("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

`${model}` is the capitalized form of the model expression, such as `booking`. Then define a public method with the name `validate${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").
                defaultMessage("Check in date must be a future date").build());
        } else if (!booking.getCheckinDate().before(booking.getCheckoutDate())) {
            messages.addMessage(new MessageBuilder().error().source("checkoutDate").
                defaultMessage("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.

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.

4.11. 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.

4.12. 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 false to prevent the transition from executing. 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;
        }
    }
}
```



Note

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.

4.13. 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 an 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.
```

4.14. 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.

4.15. 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">
```

5. Executing actions

5.1. 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.

5.2. 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 an 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-2.0.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>
```

5.3. 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>
```

5.4. 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 5.1. Action method return value to event id mappings

Method return type	Mapped Event identifier expression
<code>java.lang.String</code>	the String value
<code>java.lang.Boolean</code>	yes (for true), no (for false)
<code>java.lang.Enum</code>	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>
```

5.5. 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) {  
        ...  
    }  
    ...  
}
```

5.6. 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 equivalent 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();
        }
    }
}
```

5.7. 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-2.0.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-2.0.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>
</subflow-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. 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.

6. Flow Managed Persistence

6.1. 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.

6.2. 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-2.0.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>
```

```
</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. Securing Flows

7.1. 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.

7.2. 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.

7.3. The secured element

The secured element designates that its containing element should apply the authorization check before fully entering. This may not occur more than 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.

7.4. 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>
```

```
</bean>
```

7.5. 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 `AccessDeniedExceptions` 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>
```


8. Flow Inheritance

8.1. 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.

8.2. 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.

8.3. 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">
```

8.4. 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">
```

8.5. Inheritance Algorithm

When a child flow inherits from its 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

9. System Setup

9.1. Introduction

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

9.2. webflow-config.xsd

Web Flow provides a Spring schema that allows you to configure the system. To use this schema, include it in one of your infrastructure-layer beans files:

```
<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-2.5.xsd
    http://www.springframework.org/schema/webflow-config
    http://www.springframework.org/schema/webflow-config/spring-webflow-config-2.0.xsd">

  <!-- Setup Web Flow here -->

</beans>
```

9.3. 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:

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

FlowExecutor

Deploy a FlowExecutor, the central service for executing flows:

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

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.

9.4. 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.

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

Assigning custom flow identifiers

Specify an `id` to assign a custom registry identifier to a flow:

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

Assigning flow meta-attributes

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

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

Registering flows using a location pattern

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

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

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'`.

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

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:

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

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.

```
<!-- 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>
```

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.

```
<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" />
```

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:

```
<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="..." />
```

conversion-service

Use the `conversion-service` attribute to customize the `ConversionService` used by the Web Flow system. Converters are used to convert from one type to another when required during flow execution. The default `ConversionService` registers converters for your basic object types such as numbers, classes, and enums.

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 OGNL 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.

9.5. 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:

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

You may also configure a listener to observe only certain flows:

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

Tuning FlowExecution persistence

Use the `flow-execution-repository` element to tune flow execution persistence settings:

```
<flow-execution-repository max-executions="5" max-execution-snapshots="30" />
```

max-executions

Tune the `max-executions` attribute to place a cap on the number of flow executions that can be created per user session.

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.

10. Spring MVC Integration

10.1. 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.

10.2. 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>
```

10.3. 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`:

```
<!-- 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.

10.4. 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 `ExceptionHandler` 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"`

10.5. 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 OGNL or the Unified EL for view-to-model data binding. See the JavaDoc API of this class for more information.

10.6. 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. Spring JavaScript Quick Reference

11.1. Introduction

Spring Javascript (spring-js) is a lightweight abstraction over common JavaScript toolkits such as Dojo. It aims to provide a common client-side programming model for progressively enhancing a web page with rich widget behavior and Ajax remoting.

Use of the Spring JS API is demonstrated in the the Spring MVC + Web Flow version of the Spring Travel reference application. In addition, the JSF components provided as part of the Spring Faces library build on Spring.js.

11.2. Serving Javascript Resources

Spring JS provides a generic `ResourceServlet` to serve web resources such as JavaScript and CSS files from jar files, as well as the webapp root directory. This servlet provides a convenient way to serve Spring.js files to your pages. To deploy this servlet, declare the following in `web.xml`:

```
<!-- Serves static resource content from .jar files such as spring-js.jar -->
<servlet>
  <servlet-name>Resource Servlet</servlet-name>
  <servlet-class>org.springframework.js.resource.ResourceServlet</servlet-class>
</servlet>

<!-- Map all /resources requests to the Resource Servlet for handling -->
<servlet-mapping>
  <servlet-name>Resource Servlet</servlet-name>
  <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

11.3. 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" />" />
```

11.4. 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>
```

```
</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 the section on [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: {
</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>
```

11.5. 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 handler 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.js.ajax.AjaxUrlBasedViewResolver">
  <property name="viewClass" value="org.springframework.webflow.mvc.view.FlowAjaxTilesView"/>
</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>
```


12. JSF Integration

12.1. Introduction

Spring Faces is Spring's JSF integration module that simplifies using JSF with Spring. It lets you use the JSF UI Component Model with Spring MVC and Spring Web Flow controllers.

Spring Faces also includes a small Facelets component library that provides Ajax and client-side validation capabilities. This component library builds on Spring Javascript, a Javascript abstraction framework that integrates Dojo as the underlying UI toolkit.

12.2. Spring-centric Integration Approach

Spring Faces combines the strengths of JSF, its UI component model, with the strengths of Spring, its controller and configuration model. This brings you all the strengths of JSF without any of the weaknesses.

Spring Faces provides a powerful supplement to a number of the standard JSF facilities, including:

1. managed bean facility
2. scope management
3. event handling
4. navigation rules
5. easy modularization and packaging of views
6. cleaner URLs
7. model-level validation
8. client-side validation and UI enhancement
9. Ajax partial page updates and full navigation
10. progressive enhancement and graceful degradation

Using these features will significantly reduce the amount of configuration required in `faces-config.xml` while providing a cleaner separation between the view and controller layer and better modularization of your application's functional responsibilities. These use of these features are outlined in the sections to follow. As the majority of these features build on the flow definition language of Spring Web Flow, it is assumed that you have an understanding of the foundations presented in [Defining Flows](#) .

12.3. Configuring web.xml

The first step to using Spring Faces 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 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 Spring Faces.

```
<!-- 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>
```

When using the Spring Faces components, you also need to configure the Spring JavaScript `ResourceServlet` so that CSS and JavaScript resources may be output correctly by the components. This servlet must be mapped to `/resources/*` in order for the URL's rendered by the components to function correctly.

```
<!-- Serves static resource content from .jar files such as spring-faces.jar -->
<servlet>
  <servlet-name>Resource Servlet</servlet-name>
  <servlet-class>org.springframework.js.resource.ResourceServlet</servlet-class>
  <load-on-startup>0</load-on-startup>
</servlet>

<!-- Map all /resources requests to the Resource Servlet for handling -->
<servlet-mapping>
  <servlet-name>Resource Servlet</servlet-name>
  <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

The Spring Faces components require the use of Facelets instead of JSP, so the typical Facelets configuration must be added as well when using these components.


```

!-- 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>

```

For optimal page-loading performance, the Spring Faces component library includes a few special components: `includeStyles` and `includeScripts`. These components will eagerly load the necessary CSS stylesheets and JavaScript files at the position they are placed in your JSF view template. In accordance with the recommendations of the Yahoo Performance Guildlines, these two tags should be placed in the head section of any page that uses the Spring Faces components. For example:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
<f:view xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:sf="http://www.springframework.org/tags/faces"
  contentType="text/html" encoding="UTF-8">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <title>Spring Faces: Hotel Booking Sample Application</title>

  <sf:includeStyles />
  <sf:includeScripts />

  <ui:insert name="headIncludes"/>
</head>
...
</html>
</f:view>

```

This shows the opening of a typical Facelets XHTML layout template that uses these components to force the loading of the needed CSS and JavaScript resources at the ideal position.

The `includeStyles` component includes the necessary resources for the Dojo widget theme. By default, it includes the resources for the "tundra" theme. An alternate theme may be selected by setting the optional "theme" and "themePath" attributes on the `includeStyles` component. For example:

```

<sf:includeStyles themePath="/styles/" theme="foobar"/>

```

will try to load a CSS stylesheet at `"/styles/foobar/foobar.css"` using the Spring JavaScript ResourceServlet.

12.4. Configuring Web Flow to render JSF views

The next step is to configure Web Flow to render JSF views. To do this, in your Spring Web Flow configuration include the `faces` namespace and link in the `faces-flow-builder-services`:

```

<?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"
xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
  http://www.springframework.org/schema/webflow-config
  http://www.springframework.org/schema/webflow-config/spring-webflow-config-2.0.xsd
  http://www.springframework.org/schema/faces
  http://www.springframework.org/schema/faces/spring-faces-2.0.xsd">

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

  <!-- The registry of executable flow definitions -->
  <webflow:flow-registry id="flowRegistry" flow-builder-services="facesFlowBuilderServices" 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="facesFlowBuilderServices" />

</beans>

```

The `faces:flow-builder-services` tag also configures several other defaults appropriate for a JSF environment. Specifically, the Unified EL is configured as the default Expression Language.

See the `swf-booking-faces` reference application in the distribution for a complete working example.

12.5. Configuring faces-config.xml

The only configuration needed in `faces-config.xml` is specific to the use of Facelets. If you are using JSP and not using the Spring Faces components, you do not need to add anything specific to Spring Faces to your `faces-config.xml`

```

<faces-config>
  <application>
    <!-- Enables Facelets -->
    <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
  </application>
</faces-config>

```

12.6. Replacing the JSF Managed Bean Facility

Spring Faces allows you to completely replace the JSF managed bean facility with a combination of 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. The only available 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. Fortunately web flow provides such extended facilities.

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 Faces 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 and 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>
```

The custom `DataModel` provides some extra conveniences such as being serializable for storage beyond request scope and 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>
```

12.7. 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 definition 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 redisplay 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 definition 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 Spring Faces, 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

Spring Faces provides some special `UICommand` components that go beyond the standard JSF components by adding the ability to do Ajax-based partial view updates. These components degrade gracefully so that the flow will still be fully functional by falling back to full page refreshes if a user with a less capable browser views the page.



Note

Though the core JSF support in Spring Faces is JSF 1.1-compatible, the Spring Faces Ajax components require JSF 1.2.

Revisiting the earlier example with the paged table, you can change the "More Results" link to use an Ajax request by replacing the standard `commandButton` with the Spring Faces version (note that the Spring Faces command components use Ajax by default, but they can alternately be forced to use a normal form submit by setting `ajaxEnabled="false"` on the component):

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

This event is handled just as in the non-Ajax case with the `transition` element, but now you will add a special `render` action that specifies which portions of the component tree need to be re-rendered:

```
<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>
```

The `fragments="hotels:searchResultsFragment"` is an instruction that will be interpreted at render time, such that only the component with the JSF `clientId` "hotels:searchResultsFragment" will be rendered and returned to the client. This fragment will then be automatically replaced in the page. The `fragments` attribute can be a comma-delimited list of ids, with each id representing the root node of a subtree (meaning the root node and all of its children) to be rendered. If the "next" event is fired in a non-Ajax request (i.e., if JavaScript is disabled on the client), the `render` action will be ignored and the full page will be rendered as normal.

In addition to the Spring Faces `commandLink` component, there is a corresponding `commandButton` component with the same functionality. There is also a special `ajaxEvent` component that will raise a JSF action even in response to any client-side DOM event. See the Spring Faces tag library docs for full details.

An additional built-in feature when using the Spring Faces Ajax components is the ability to have the response rendered inside a rich modal popup widget by setting `popup="true"` on a `view-state`.

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

```
<render fragments="hotelSearchFragment" />
</on-entry>
<transition on="search" to="reviewHotels">
  <evaluate expression="searchCriteria.resetPage()" />
</transition>
</view-state>
```

If the "changeSearchCriteria" view-state is reached as the result of an Ajax-request, the result will be rendered into a rich popup. If JavaScript is unavailable, the request will be processed with a full browser refresh, and the "changeSearchCriteria" view will be rendered as normal.

12.8. Enhancing The User Experience With Rich Web Forms

JSF and Web Flow combine to provide an extensive server-side validation model for your web application, but excessive roundtrips to the server to execute this validation and return error messages can be a tedious experience for your users. Spring Faces provides a number of client-side rich validation controls that can enhance the user experience by applying simple validations that give immediate feedback. Some simple examples are illustrated below. See the Spring Faces taglib docs for a complete tag reference.

Validating a Text Field

Simple client-side text validation can be applied with the `clientTextValidator` component:

```
<sf:clientTextValidator required="true">
  <h:inputText id="creditCardName" value="#{booking.creditCardName}" required="true"/>
</sf:clientTextValidator>
```

This will apply client-side required validation to the child `inputText` component, giving the user a clear indicator if the field is left blank.

Validating a Numeric Field

Simple client-side numeric validation can be applied with the `clientNumberValidator` component:

```
<sf:clientTextValidator required="true" regExp="[0-9]{16}"
  invalidMessage="A 16-digit credit card number is required.">
  <h:inputText id="creditCard" value="#{booking.creditCard}" required="true"/>
</sf:clientTextValidator>
```

This will apply client-side validation to the child `inputText` component, giving the user a clear indicator if the field is left blank, is not numeric, or does not match the given regular expression.

Validating a Date Field

Simple client-side date validation with a rich calendar popup can be applied with the `clientDateValidator` component:

```
<sf:clientDateValidator required="true" >
  <h:inputText id="checkinDate" value="#{booking.checkinDate}" required="true">
    <f:convertDateTime pattern="yyyy-MM-dd" timeZone="EST"/>
  </h:inputText>
</sf:clientDateValidator>
```

This will apply client-side validation to the child `inputText` component, giving the user a clear indicator if the field is left blank or is not a valid date.

Preventing an Invalid Form Submission

The `validateAllOnClick` component can be used to intercept the "onclick" event of a child component and suppress the event if all client-side validations do not pass.

```
<sf:validateAllOnClick>
  <sf:commandButton id="proceed" action="proceed" processIds="" value="Proceed"/>&#160;
</sf:validateAllOnClick>
```

This will prevent the form from being submitted when the user clicks the "proceed" button if the form is invalid. When the validations are executed, the user is given clear and immediate indicators of the problems that need to be corrected.

12.9. Third-Party Component Library Integration

Spring Faces 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 Spring Faces 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.) In some cases, a deeper level of integration can be achieved by configuring special flow services that are "aware" of a particular component library, and these will be noted in the examples to follow.

Rich Faces Integration

To use the Rich Faces component library with Spring Faces, the following filter configuration is needed in `web.xml` (in addition to the typical Spring Faces configuration):

```
<filter>
  <display-name>RichFaces Filter</display-name>
```

```

    <filter-name>richfaces</filter-name>
    <filter-class>org.ajax4jsf.Filter</filter-class>
</filter>

<filter-mapping>
    <filter-name>richfaces</filter-name>
    <servlet-name>Spring Web MVC Dispatcher Servlet</servlet-name>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
</filter-mapping>

```

For deeper integration (including the ability to have a view with combined use of the Spring Faces Ajax components and Rich Faces Ajax components), configure the RichFacesAjaxHandler on your FlowController:

```

<bean id="flowController" class="org.springframework.webflow.mvc.servlet.FlowController">
    <property name="flowExecutor" ref="flowExecutor" />
    <property name="ajaxHandler">
        <bean class="org.springframework.faces.richfaces.RichFacesAjaxHandler"/>
    </property>
</bean>

```

RichFaces Ajax components can be used in conjunction with the render tag to render partial fragments on an Ajax request. Instead of embedding the ids of the components to be re-rendered directly in the view template (as you traditionally do with Rich Faces), you can bind the reRender attribute of a RichFaces Ajax component to a special flowRenderFragments EL variable. For example, in your view template you can have a fragment that you would potentially like to re-render in response to a particular event:

```

<h:form id="hotels">
    <a4j:outputPanel id="searchResultsFragment">
        <h:outputText id="noHotelsText" value="No Hotels Found" rendered="#{hotels.rowCount == 0}"/>
        <h:dataTable id="hotels" styleClass="summary" value="#{hotels}" var="hotel" rendered="#{hotels.rowCount > 0}">
            <h:column>
                <f:facet name="header">Name</f:facet>
                #{hotel.name}
            </h:column>
            <h:column>
                <f:facet name="header">Address</f:facet>
                #{hotel.address}
            </h:column>
        </h:dataTable>
    </a4j:outputPanel>
</h:form>

```

then a RichFaces Ajax commandLink to fire the event:

```

<a4j:commandLink id="nextPageLink" value="More Results" action="next" reRender="#{flowRenderFragments}" />

```

and then in your flow definition a transition to handle the event:

```

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

```

Apache MyFaces Trinidad Integration

The Apache MyFaces Trinidad library has been tested with the Spring Faces integration and proven to fit in nicely. Deeper integration to allow the Trinidad components and Spring Faces components to play well together has not yet been attempted, but Trinidad provides a pretty thorough solution on its own when used in conjunction with the Spring Faces integration layer.

NOTE - An AjaxHandler implementation for Trinidad is not currently provided out-of-the-box with Spring Faces. In order to fully integrate with Trinidad's PPR functionality, a custom implementation should be provided. An community-provided partial example can be found here: [SWF-1160](#)

Typical Trinidad + Spring Faces configuration is as follows in web.xml (in addition to the typical Spring Faces configuration):

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>server</param-value>
</context-param>

<context-param>
  <param-name>
    org.apache.myfaces.trinidad.CHANGE_PERSISTENCE
  </param-name>
  <param-value>session</param-value>
</context-param>

<context-param>
  <param-name>
    org.apache.myfaces.trinidad.ENABLE_QUIRKS_MODE
  </param-name>
  <param-value>>false</param-value>
</context-param>

<filter>
  <filter-name>Trinidad Filter</filter-name>
  <filter-class>
    org.apache.myfaces.trinidad.webapp.TrinidadFilter
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>Trinidad Filter</filter-name>
  <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
</filter-mapping>

<servlet>
  <servlet-name>Trinidad Resource Servlet</servlet-name>
  <servlet-class>
    org.apache.myfaces.trinidad.webapp.ResourceServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>resources</servlet-name>
  <url-pattern>/adf/*</url-pattern>
</servlet-mapping>
```


13. Portlet Integration

13.1. Introduction

This chapter shows how to use Web Flow in a Portlet environment. Web Flow has full support for JSR-168 portlets. The `booking-portlet-mvc` sample application is a good reference for using Web Flow within a portlet. This application is a simplified travel site that allows users to search for and book hotel rooms.

13.2. Configuring `web.xml` and `portlet.xml`

The configuration for a portlet depends on the portlet container used. The sample applications, included with Web Flow, are both configured to use [Apache Pluto](#), the JSR-168 reference implementation.

In general, the configuration requires adding a servlet mapping in the `web.xml` file to dispatch request to the portlet container.

```
<servlet>
  <servlet-name>swf-booking-mvc</servlet-name>
  <servlet-class>org.apache.pluto.core.PortletServlet</servlet-class>
  <init-param>
    <param-name>portlet-name</param-name>
    <param-value>swf-booking-mvc</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>swf-booking-mvc</servlet-name>
  <url-pattern>/PlutoInvoker/swf-booking-mvc</url-pattern>
</servlet-mapping>
```

The `portlet.xml` configuration is a standard portlet configuration. The `portlet-class` needs to be set along with a pair of `init-params`. Setting the `expiration-cache` to 0 is recommended to force Web Flow to always render a fresh view.

```
<portlet>
  ...
  <portlet-class>org.springframework.web.portlet.DispatcherPortlet</portlet-class>
  <init-param>
    <name>contextConfigLocation</name>
    <value>/WEB-INF/web-application-config.xml</value>
  </init-param>
  <init-param>
    <name>viewRendererUrl</name>
    <value>/WEB-INF/servlet/view</value>
  </init-param>
  <expiration-cache>0</expiration-cache>
  ...
</portlet>
```

13.3. Configuring Spring

Flow Handlers

The only supported mechanism for bridging a portlet request to Web Flow is a `FlowHandler`. The `PortletFlowController` used in Web Flow 1.0 is no longer supported.

The flow handler, similar to the servlet flow handler, provides hooks that can:

- select the flow to execute
- pass input parameters to the flow on initialization
- handle the flow execution outcome
- handle exceptions

The `AbstractFlowHandler` class is an implementation of `FlowHandler` that provides default implementations for these hooks.

In a portlet environment the targeted flow id can not be inferred from the URL and must be defined explicitly in the handler.

```
public class ViewFlowHandler extends AbstractFlowHandler {  
    public String getFlowId() {  
        return "view";  
    }  
}
```

Handler Mappings

Spring Portlet MVC provides a rich set of methods to map portlet requests. Complete documentation is available in the [Spring Reference Documentation](#).

The `booking-portlet-mvc` sample application uses a `PortletModeHandlerMapping` to map portlet requests. The sample application only supports view mode, but support for other portlet modes is available. Other modes can be added and point to the same flow as view mode, or any other flow.

```
<bean id="portletModeHandlerMapping"  
    class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">  
    <property name="portletModeMap">  
        <map>  
            <entry key="view">  
                <bean class="org.springframework.webflow.samples.booking.ViewFlowHandler" />  
            </entry>  
        </map>  
    </property>  
</bean>
```

Flow Handler Adapter

A `FlowHandlerAdapter` converts the handler mappings to the flow handlers. The flow executor is required as a constructor argument.

```
<bean id="flowHandlerAdapter"
      class="org.springframework.webflow.mvc.portlet.FlowHandlerAdapter">
  <constructor-arg ref="flowExecutor" />
</bean>
```

13.4. Portlet Views

In order to facilitate view rendering, a `ViewRendererServlet` must be added to the `web.xml` file. This servlet is not invoked directly, but it is used by Web Flow to render views in a portlet environment.

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

<servlet-mapping>
  <servlet-name>ViewRendererServlet</servlet-name>
  <url-pattern>/WEB-INF/servlet/view</url-pattern>
</servlet-mapping>
```

13.5. Portlet Modes and Window States

Window State

The Portlet API defined three window states: normal, minimized and maximized. The portlet implementation must decide what to render for each of these window states. Web Flow exposes the string value of the window state under `portletWindowState` via the request map on the external context.

```
requestContext.getExternalContext().getRequestMap().get("portletWindowState");
```

```
externalContext.getRequestMap().portletWindowState
```

Portlet Mode

The Portlet API defined three portlet modes: view, edit and help. The portlet implementation must decide what to render for each of these modes. Web Flow exposes the string value of the portlet mode under `portletMode` via the request map on the external context.

```
requestContext.getExternalContext().getRequestMap().get("portletMode");
```

```
externalContext.requestMap.portletMode
```

13.6. Issues in a Portlet Environment

Redirects

The Portlet API only allows redirects to be requested from an action request. Because views are rendered on the render request, views and view-states cannot trigger a redirect.

The `externalRedirect: view` prefix is a convenience for Servlet based flows. An `IllegalStateException` is thrown if a redirect is requested from a render request.

end-state redirects can be achieved by implementing `FlowHandler.handleExecutionOutcome`. This callback provides the `ActionResponse` object which supports redirects.

Switching Portlet Modes

The portlet container passes the execution key from the previous flow when switching to a new mode. Even if the mode is mapped to a different `FlowHandler` the flow execution will resume the previous execution. You may switch the mode programatically in your `FlowHandler` after ending a flow in an `ActionRequest`.

One way to start a new flow is to create a URL targeting the mode without the execution key.

Portlets and JSF

Web Flow supports JSF as the view technology for a portlet. However, a jsf-portlet bridge (JSR-301) must be provided. At the time of this writing, no feature complete jsf-portlet bridge exists. Some of the existing bridge implementations may appear to work, however, side effect may occur.

JSF portlets are considered experimental at this time.

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. Your 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;
}
```


15. Upgrading from 1.0

15.1. Introduction

This chapter shows you how to upgrade existing Web Flow 1 application to Web Flow 2.

15.2. Flow Definition Language

The core concepts behind the flow definition language have not changed between Web Flow 1 and 2. However, some of the element and attribute names have changed. These changes allow for the language to be both more concise and expressive. A complete list of [mapping changes](#) is available as an appendix.

Flow Definition Updater Tool

An automated tool is available to aid in the conversion of existing 1.x flows to the new 2.x style. The tool will convert all the old tag names to their new equivalents, if needed. While the tool will make a best effort attempt at conversion, there is not a one-to-one mapping for all version 1 concepts. If the tool was unable to convert a portion of the flow, it will be marked with a WARNING comment in the resulting flow.

The conversion tool requires spring-webflow.jar, spring-core.jar and an XSLT 1.0 engine. [Saxon 6.5.5](#) is recommended.

The tool can be run from the command line with the following command. Required libraries must be available on the classpath. The source must be a single flow to convert. The resulting converted flow will be sent to standard output.

```
java org.springframework.webflow.upgrade.WebFlowUpgrader flow-to-upgrade.xml
```

Flow Definition Updater Tool Warnings

argument parameter-type no longer supported

Bean actions have been deprecated in favor of EL based evaluate expressions. The EL expression is able to accept method parameters directly, so there is no longer a need for the argument tag. A side effect of this change is that method arguments must be of the correct type before invoking the action.

inline-flow is no longer supported

Inline flows are no longer supported. The contents of the inline flow must be moved into a new

top-level flow. The inline flow's content has been converted for your convenience.

mapping target-collection is no longer supported

Output mappings can no longer add an item to a collection. Only assignment is supported.

var bean is no longer supported

The var bean attribute is no longer needed. All spring beans can be resolved via EL.

var scope is no longer supported

The var element will place all variable into flow scope. Conversation scope was previously allowed.

EL Expressions

EL expressions are used heavily throughout the flow definition language. Many of the attributes that appear to be plain text are actually interpreted as EL. The standard EL delimiters (either `${ }` or `#{ }`) are not necessary and will often cause an exception if they are included.

EL delimiters should be removed where necessary by the updater tool.

15.3. Web Flow Configuration

In Web Flow 1 there were two options available for configuring Web Flow, one using standard spring bean XML and the other using the `webflow-config-1.0` schema. The schema configuration option simplifies the configuration process by keeping long internal class names hidden and enabling contextual auto-complete. The schema configuration option is the only way to configure Web Flow 2.

Web Flow Bean Configuration

The `FactoryBean` bean XML configuration method used in Web Flow 1 is no longer supported. The schema configuration method should be used instead. In particular beans defining `FlowExecutorFactoryBean` and `XmlFlowRegistryFactoryBean` should be updated. Continue reading Web Flow Schema Configuration for details.

Web Flow Schema Configuration

The `webflow-config` configuration schema has also changed slightly from version 1 to 2. The simplest way to update your application is modify the version of the schema to 2.0 then fix any errors in a schema aware XML editor. The most common change is add 'flow-' to the

beginning of the elements defined by the schema.

```
<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-2.5.xsd
         http://www.springframework.org/schema/webflow-config
         http://www.springframework.org/schema/webflow-config/spring-webflow-config-2.0.xsd">
```

flow-executor

The flow executor is the core Web Flow configuration element. This element replaces previous `FlowExecutorFactoryBean` bean definitions.

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

flow-execution-listeners

Flow execution listeners are also defined in the flow executor. Listeners are defined using standard bean definitions and added by reference.

```
<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" />
```

flow-registry

The flow-registry contains a set of flow-locations. Every flow definition used by Web Flow must be added to the registry. This element replaces previous `XmlFlowRegistryFactoryBean` bean definitions.

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

Flow Controller

The package name for flow controllers has changed from `org.springframework.webflow.executor.mvc.FlowController` and is now `org.springframework.webflow.mvc.servlet.FlowController` for Servlet MVC requests. The portlet flow controller `org.springframework.webflow.executor.mvc.PortletFlowController` has been replaced by a flow handler adapter available at `org.springframework.webflow.mvc.portlet.FlowHandlerAdapter`. They

will need to be updated in the bean definitions.

Flow URL Handler

The default URL handler has changed in Web Flow 2. The flow identifier is now derived from the URL rather than passed explicitly. In order to maintain comparability with existing views and URL structures a `WebFlow1FlowUrlHandler` is available.

```
<bean name="/pos.htm" class="org.springframework.webflow.mvc.servlet.FlowController">
  <property name="flowExecutor" ref="flowExecutor" />
  <property name="flowUrlHandler">
    <bean class="org.springframework.webflow.context.servlet.WebFlow1FlowUrlHandler" />
  </property>
</bean>
```

View Resolution

Web Flow 2 by default will both select and render views. View were previously selected by Web Flow 1 and then rendered by an external view resolver.

In order for version 1 flows to work in Web Flow 2 the default view resolver must be overridden. A common use case is to use [Apache Tiles](#) for view resolution. The following configuration will replace the default view resolver with a Tiles view resolver. The `tilesViewResolver` in this example can be replaced with any other view resolver.

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

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

<bean id="viewFactoryCreator" class="org.springframework.webflow.mvc.builder.MvcViewFactoryCreator">
  <property name="viewResolvers" ref="tilesViewResolver" />
</bean>

<bean id="tilesViewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.tiles.TilesJstlView" />
</bean>

<bean class="org.springframework.web.servlet.view.tiles.TilesConfigurer">
  <property name="definitions" value="/WEB-INF/tiles-def.xml" />
</bean>
```

15.4. New Web Flow Concepts

Automatic Model Binding

Web Flow 1 required Spring MVC based flows to manually call `FormAction` methods, notably: `setupForm`, `bindAndValidate` to process form views. Web Flow 2 now provides automatic model setup and binding using the `model` attribute for view-states. Please see the [Binding to a Model](#) section for details.

OGNL vs EL

Web Flow 1 used OGNL exclusively for expressions within the flow definitions. Web Flow 2 adds support for Unified EL. Unified EL is used when it is available, OGNL will continue to be used when a Unified EL implementation is not available. Please see the [Expression Language](#) chapter for details.

Flash Scope

Flash scope in Web Flow 1 lived across the current request and into the next request. This was conceptually similar to Web Flow 2's view scope concept, but the semantics were not as well defined. In Web Flow 2, flash scope is cleared after every view render. This makes flashScope semantics in Web Flow consistent with other web frameworks.

Spring Faces

Web Flow 2 offers significantly improved integration with JavaServerFaces. Please see the [JSF Integration](#) chapter for details.

External Redirects

External redirects in Web Flow 1 were always considered context relative. In Web Flow 2, if the redirect URL begins with a slash, it is considered servlet-relative instead of context-relative. URLs without a leading slash are still context relative.

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

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

SWF 1.0		SWF 2.0		Comments
	else		else	
	<i>import</i>		<i>bean-import</i>	
	resource		resource	
	<i>inline-flow</i>	*		convert to new top-level flow
	id		*	
	<i>input-attribute</i>		<i>input</i>	
	name		name	
	scope		*	prefix name with scope <input name="flowScope.foo" />
	required		required	
	*		type	
	*		value	
	<i>input-mapper</i>	*		inputs can be in flows and subflows directly
	<i>mapping</i>		<i>input or output</i>	
	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	
	<i>method-argument</i>	*		use <evaluate expression="func(arg1, arg2, ...)"/>
	<i>method-result</i>	*		use <evaluate result="..." />
	name		*	
	scope		*	
	<i>output-attribute</i>		<i>output</i>	
	name		name	
	scope		*	prefix name with scope <output name="flowScope.foo" />

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

SWF 1.0		SWF 2.0		Comments
	class		class	
	scope		*	always flow scope
	bean		*	all Spring beans can be resolved with EL
	<i>view-state</i>		<i>view-state</i>	
	id		id	
	view		view	
	*		parent	
	*		redirect	
	*		popup	
	*		model	
*	*		history	
*			<i>persistence-context</i>	
			<i>render</i>	
*	*		fragments	
			<i>secured</i>	
	*		attributes	
	*		match	

