The Definitive Guide to Spring Web Flow

Copyright © 2008 by Erwin Vervaet

ISBN-13 (pbk): 978-1-4302-1624-7

ISBN-13 (electronic): 978-1-4302-1625-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

SpringSource is the company behind Spring, the de facto standard in enterprise Java. SpringSource is a leading provider of enterprise Java infrastructure software, and delivers enterprise class software, support, and services to help organizations utilize Spring. The open source-based Spring Portfolio is a comprehensive enterprise application framework designed on long-standing themes of simplicity and power. With more than five million downloads to date, Spring has become an integral part of the enterprise application infrastructure at organizations worldwide. For more information visit: www.springsource.com.

Lead Editor: Steve Anglin
Technical Reviewers: Keith Donald, Karl David Moore
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Senior Project Manager: Kylie Johnston
Copy Editor: Heather Lang
Associate Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor: Pat Christenson
Proofreader: Lisa Hamilton
Indexer: Becky Hornyak
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com.

CHAPTER 7

Driving Flow Executions

The previous chapters explained the basic building blocks that make up Spring Web Flow. You learned about flow definitions and how you can use them to define the navigational rules of a use case. Using its advanced execution management system, Spring Web Flow automatically enforces those navigational rules and controls state management and cleanup. A flow executor ties all the building blocks together:

- It loads flow definitions eligible for execution from a flow definition registry.
- It creates executions for those definitions using a flow execution factory, abstracting away the underlying execution engine.
- It persists ongoing flow executions in between individual requests with the help of a flow execution repository.

In essence, a flow executor is a high-level facade for Spring Web Flow. Since Spring Web Flow is not a complete web MVC framework, but instead just a controller component for such a framework, it needs to be integrated into a hosting framework. In practice, this integration makes a flow executor available to the hosting framework. Spring Web Flow 1 integrates with several well-known web MVC frameworks, namely:

- Spring Web MVC
- Spring Portlet MVC
- Struts
- JavaServer Faces (JSF)

INTEGRATION WITH OTHER FRAMEWORKS

This chapter discusses only the host framework integrations that ship with Spring Web Flow. Spring Web Flow has also been integrated with other frameworks, notably Struts 2 and Grails. For more details, consult the documentation of those frameworks.

The Spring Web Flow plug-in that integrates Spring Web Flow into the Apache Struts 2 framework can be found at `http://cwiki.apache.org/S2PLUGINS/spring-webflow-plugin.html`.

Grails, a rapid web application development framework heavily based on the convention-over-configuration concept and the Groovy language, completely encapsulates Spring Web Flow as a flow engine. For instance, Grails provides its own Groovy-based flow definition syntax. By doing that, Grails can offer users a very integrated and consistent development experience, while still leveraging Spring Web Flow's navigational control and state management. For more information about Grails, visit `http://www.grails.org`.

Chapter 11 will illustrate how to build your own integration code, using a `FlowServlet` as an example.

Before diving into actual integration details, let's first look at some general topics related to this subject.

Flow Executor Integration

Code that integrates Spring Web Flow into a particular hosting framework acts as a *front-end* to a flow executor. As such, it needs to fulfill two responsibilities:

- *Extracting flow executor method arguments from the request*: Key arguments here are the flow ID, the flow execution key, and the event to signal.
- *Exposing values such as the flow execution key to the view that will be rendered*: These values are contained in the `ResponseInstruction` returned by the `FlowExecutor`. Making them available to the view ensures that the view can embed those values into later requests.

■**Note** Recall that Spring Web Flow will encapsulate a framework-specific request, such as an `HttpServletRequest` in a typical Java web application, inside an `ExternalContext` object. This decouples Spring Web Flow from details specific to the hosting framework or protocols used. Refer to the "Flow Executions" section in Chapter 4 for more details.

Clearly, both responsibilities need to be in sync: values need to be exposed in such a way that they can be extracted again on a subsequent request. To enforce this, Spring Web Flow defines the concept of a `FlowExecutorArgumentHandler`, which combines the responsibilities of a `FlowExecutorArgumentExtractor` and a `FlowExecutorArgumentExposer`, as shown in Figure 7-1. As the name suggests, a `FlowExecutorArgumentHandler` will handle extracting flow executor arguments from the external context and exposing those arguments again later on.

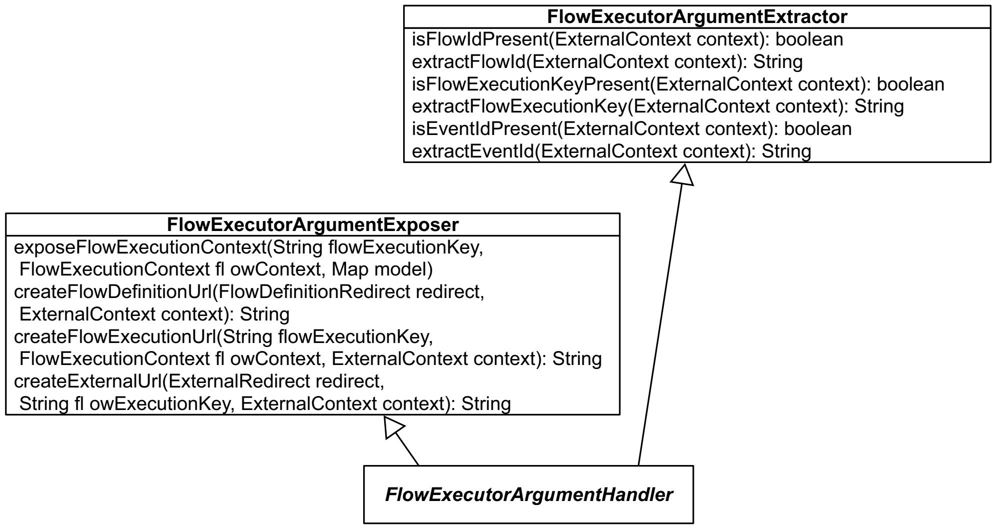
FlowExecutorArgumentExtractor
isFlowIdPresent(ExternalContext context): boolean
extractFlowId(ExternalContext context): String
isFlowExecutionKeyPresent(ExternalContext context): boolean
extractFlowExecutionKey(ExternalContext context): String
isEventIdPresent(ExternalContext context): boolean
extractEventId(ExternalContext context): String
FlowExecutorArgumentExposer
exposeFlowExecutionContext(String flowExecutionKey, FlowExecutionContext fl owContext, Map model)
createFlowDefinitionUrl(FlowDefinitionRedirect redirect, ExternalContext context): String
createFlowExecutionUrl(String flowExecutionKey, FlowExecutionContext fl owContext, ExternalContext context): String
createExternalUrl(ExternalRedirect redirect, String fl owExecutionKey, ExternalContext context): String
FlowExecutorArgumentHandler

Figure 7-1. *FlowExecutorArgumentHandler class diagram*

Spring Web Flow provides three `FlowExecutorArgumentHandler` implementations out of the box. The default argument handler is the `RequestParameterFlowExecutorArgumentHandler`, which extracts and exposes flow executor argument values as HTTP request parameters. There is also a `RequestPathFlowExecutorArgumentHandler`, capable of extracting and exposing flow executor arguments in the request path, giving REST-style URLs. Finally, the `FlowIdMappingArgumentHandlerWrapper` simply wraps another `FlowExecutorArgumentHandler`, applying a flow ID mapping along the way (for example, `enterPayment` is mapped to `enterPayment-flow`). This mapping allows you to control the flow IDs that are visible in the URLs and pages of your web application. You can, of course, also implement your own `FlowExecutorArgumentHandler`.

All host framework integrations discussed in the "Host Framework Integrations" section are essentially flow executor front-ends performing the following simple steps:

1. They wrap the incoming request in an `ExternalContext` implementation, abstracting away framework and protocol details.

2. Using an argument handler, they extract flow executor arguments from the request (which has been wrapped in an `ExternalContext`).

3. They call into the flow executor, passing in all extracted arguments as well as the `ExternalContext` object itself.

4. They convert the `ResponseInstruction` returned by the flow executor into an appropriate response (for example, rendering a view or issuing a redirect) using hosting framework specific techniques. Flow executor arguments for a subsequent request are exposed to the view using the same argument handler used to extract values from the request.

It clearly follows that a flow executor front-end needs three components to function properly: a `FlowExecutor`, a `FlowExecutorArgumentHandler`, and an appropriate `ExternalContext` implementation. You will see this pattern appear again and again in the "Host Framework Integrations" section, which discusses actual integration details.

Before coming to that, let's first take a deeper look at developing views that participate in a web flow. What model data will Spring Web Flow expose to views, and how do you build requests from which Spring Web Flow can extract the necessary values?

Spring Web Flow View Development

Since Spring Web Flow is essentially a controller component in an MVC framework, it does not dictate the view technology used. Any view technology supported by the hosting framework can be used, for instance, JavaServer Pages (JSP), Velocity (`http://velocity.apache.org`), or FreeMarker (`http://freemarker.sourceforge.net`) templates, XSLT style sheets, and so on. However, there are a few common Spring Web Flow–specific things to point out.

Model Data

As an MVC controller, Spring Web Flow prepares model data that will be rendered by the view it selects. Making the model data available for view rendering is the responsibility of the hosting framework. When using JSP, for instance, the model is typically made available to the view as request attributes. This might be different for other view technologies though. For instance, in the case of XSLT style sheets, the model will need to be turned into an XML document before being transformed.

The model data prepared by Spring Web Flow consists largely of application-specific information supplemented with some extra attributes exposed using the `FlowExecutorArgumentHandler`, this time in its role of `FlowExecutorArgumentExposer`. Overall, Spring Web Flow will make the following information available in the model:

- *The union of all flow execution scopes (request, flash, flow, and conversation scopes)*: This essentially exposes all application data managed by the flow to the view for rendering. As a developer, you just need to put objects into one of the flow execution scopes, and Spring Web Flow will make sure the view has access to them.

  If there are name clashes between objects in the different scopes, the request scope takes precedence over the flash scope, which takes precedence over the flow scope, which, in turn, takes precedence over the conversation scope. For instance, if there is an object indexed as `foo` in both the flash and conversation scopes, the `foo` object from the flash scope will end up in the model.

- *The unique key of the flow execution rendering the view*: This key is assigned by the flow execution repository and placed in the model as `flowExecutionKey`. View developers can use this value to submit the flow execution key back to the server on a next request.

- *The flow execution context*: This object implementing the `FlowExecutionContext` interface is available in the model as `flowExecutionContext`. View developers can use the flow execution context to learn about the flow execution that is rendering the view. For instance, using `FlowExecutionContext.getActiveSession().isRoot()`, the view can decide not to display a Back button if the flow is running as a top-level flow, instead of a subflow.

To make this a bit more tangible, let's look at a few examples of how you would access this model data in JSP. Most of these examples are taken from the Spring Bank sample application. Similar techniques can be used when using other view technologies.

At a very basic level, a JSP page can use Java scriptlets to access model information as request attributes:

```
<%@ page import="com.ervacon.springbank.domain.Payment" %>
<td valign="top">
  <%=((Payment)request.getAttribute("payment")).getCreditAccount(). \
    getNumber() %>
</td>

<input type="hidden" name="_flowExecutionKey"
  value="<%=request.getAttribute("flowExecutionKey") %>"/>

<%@ page
  import="org.springframework.webflow.execution.FlowExecutionContext" %>
<%
  FlowExecutionContext flowExecutionContext =
    (FlowExecutionContext)request.getAttribute("flowExecutionContext");
  if (!flowExecutionContext.getActiveSession().isRoot()) {
%>
  <input type="submit" name="_eventId_back" value="Back"/>
<%
  }
%>
```

A servlet can, of course, do something similar. With the advent of the JSP Standard Tag Library (JSTL), it is no longer necessary to pollute your JSP pages with bloated Java scriptlets like those shown in the preceding example. Instead, you can simply use the JSTL and its expression language:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<td valign="top">
  <c:out value="${payment.creditAccount.number} "/>
</td>

<input type="hidden" name="_flowExecutionKey"
  value="<c:out value="${flowExecutionKey} "/>"/>

<c:if test="${!flowExecutionContext.activeSession.root} ">
  <input type="submit" name="_eventId_back" value="Back"/>
</c:if>
```

This avoids the need for type casts and null checks, which are necessary in the Java scriptlets, making the overall page much cleaner and more readable. JSP 2.0 directly supports the JSTL expression language, making it possible to leave out the `<c:out>` tags:

```
<td valign="top">
  ${payment.creditAccount.number}
</td>

<input type="hidden" name="_flowExecutionKey"
  value="${flowExecutionKey} "/>

<c:if test="${!flowExecutionContext.activeSession.root} ">
  <input type="submit" name="_eventId_back" value="Back"/>
</c:if>
```

You still need the JSTL in some cases, for instance, to do "if" tests or to properly escape text (and avoid cross-site scripting attacks).

■ **Note** Notice that all flow execution scopes are merged together to form the model. As a result, you do not need to specify `flashScope`, `flowScope`, or any of the other scopes like you would in an OGNL expression inside a flow definition. Instead, you directly access the object using its key in the model, for instance, `payment` in this example: `${payment.creditAccount.number}`. This avoids unnecessary coupling between the view implementations and Spring Web Flow.

Next to rendering model data, views also have the important responsibility of constructing requests that lead back into the flow execution.

Building Requests

Previous chapters already identified the three types of requests recognized by Spring Web Flow, and Figure 6-1 clarified the algorithm Spring Web Flow uses to classify a request. Let's look at each request type in a bit more detail, illustrating how to build such a request in JSP.

The `FlowExecutorArgumentHandler`, this time in its capacity of `FlowExecutorArgumentExtractor`, of course influences the way you need to build up request URLs. When using the `RequestParameterFlowExecutorArgumentHandler`, values will need to be included in the request as request parameters, while they are part of the request path when using the `RequestPathFlowExecutorArgumentHandler`.

I'll start by looking at Spring Web Flow's default configuration, using request parameters and the `RequestParameterFlowExecutorArgumentHandler`.

■ **Note** Unless noted otherwise, it is assumed that the `/flows.html` URI maps to a Spring Web Flow controller.

Launching Flows

A request coming into Spring Web Flow that does not contain a `_flowExecutionKey` request parameter will be seen as a request launching a new flow execution. The ID of the flow to launch is specified using the `_flowId` request parameter. If no `_flowId` request parameter is found, the configured default flow ID is used or, if the default ID is lacking, an error is generated.

■ **Tip** The `FlowExecutorArgumentHandler` class has a `defaultFlowId` property that you can use to specify the ID of the flow to launch if no other flow ID can be extracted from the request. More specific configuration details will follow in the "Host Framework Integrations" section.

For instance, the HTML anchor rendered by the following JSP fragment launches the `enterPayment-flow`:

```
<a href="<c:url value="/flows.html?_flowId=enterPayment-flow"/>">
  <fmt:message key="enterPayment"/>
</a>
```

This produces the following HTML:

```
<a href="/springbank/flows.html?_flowId=enterPayment-flow">
  Enter Payment
</a>
```

Similarly, you can launch a flow using an HTML form:

```
<form action="<c:url value="/flows.html"/>">
  <input type="hidden" name="_flowId" value="enterPayment-flow"/>
  <input type="submit" value="<fmt:message key="enterPayment"/>"/>
</form>
```

Tip All request parameters contained in the request will also be provided as input to the launching flow, making it easy to launch a flow and pass some input to it.

Signaling Events

A request that resumes a flow execution and signals an event needs to contain both a `_flowExecutionKey` and an `_eventId` request parameter.

The following example shows an HTML anchor that submits the `next` event, along with a selected debit account number:

```
<c:url value="/flows.html" var="nextUrl">
  <c:param name="_flowExecutionKey">${flowExecutionKey} </c:param>
  <c:param name="_eventId">next</c:param>
  <c:param name="debitAccount">${account.number} </c:param>
</c:url>
<a href="${nextUrl} ">${account.number} </a>
```

Notice how the `flowExecutionKey` is accessed in the model using a simple JSP 2.0 expression. The resulting HTML looks like this:

```
<a href="/springbank/flows.html?_flowExecutionKey=_c6F..._kD8... \
  &_eventId=next&debitAccount=SpringBank-0">SpringBank-0</a>
```

Of course, you can also submit events using HTML forms:

```
<form action="<c:url value="/flows.html"/>">
  <input type="hidden" name="_flowExecutionKey" value="${flowExecutionKey} "/>
  <input type="hidden" name="_eventId" value="next"/>
  <input type="hidden" name="debitAccount" value="${account.number} "/>
  <input type="submit" value="${account.number} "/>
</form>
```

The preceding form uses a separate hidden field to explicitly submit the `_eventId` request parameter. Spring Web Flow also allows you to embed the event ID in the name of a *submit* field, like so:

```
<form action="<c:url value="/flows.html"/>">
  <input type="hidden" name="_flowExecutionKey" value="${flowExecutionKey} "/>
  <input type="hidden" name="debitAccount" value="${account.number} "/>
  <input type="submit" name="_eventId_next"
    value="${account.number} "/>
</form>
```

This technique is particularly useful if you have a single HTML form with multiple buttons that each signal a different event. Finally, Spring Web Flow will also recognize events submitted by image buttons:

```
<form action="<c:url value="/flows.html"/>">
  <input type="hidden" name="_flowExecutionKey" value="${flowExecutionKey} "/>
  <input type="hidden" name="debitAccount" value="${account.number} "/>
  <input type="image" src="images/account.png"
    name="_eventId_next" value="${account.number} "/>
</form>
```

Refreshing Flow Executions

A request that contains only a `_flowExecutionKey` request parameter, but no `_eventId`, will trigger a flow execution refresh. This was explained in detail in the "Request Life Cycle" section of Chapter 4.

Note Recall that when using "always redirect on pause," Spring Web Flow will automatically generate a redirect causing a flow execution refresh.

Although this is not generally useful, the following HTML anchor will trigger a flow execution refresh:

```
<a href="<c:url
  value="/flows.html?_flowExecutionKey=${flowExecutionKey} "/>">
  Refresh
</a>
```

The resulting HTML is simply this:

```
<a href="/springbank/flows.html?_flowExecutionKey=_c6F..._kD8...">
  Refresh
</a>
```

It goes without saying that you can also use an HTML form to trigger a refresh:

```
<form action="<c:url value="/flows.html"/>">
  <input type="hidden" name="_flowExecutionKey"
    value="${flowExecutionKey} "/>
  <input type="submit" value="Refresh"/>
</form>
```

■**Tip** Although you can submit extra request parameters with a flow execution refresh request, Spring Web Flow will not automatically process them. However, you can still use them from inside a view state render action for instance.

REST-style Requests

The examples shown in the previous sections use HTTP request parameters to submit the `_flowId`, `_flowExecutionKey`, and `_eventId` values. Using the `RequestPathFlowExecutorArgumentHandler`, Spring Web Flow also allows you to use URLs that follow the REST style by embedding the `_flowId` and `_flowExecutionKey` values inside the path of the resource identified by the URL.

■**Tip** The `RequestPathFlowExecutorArgumentHandler` always falls back to request parameters if it cannot find the necessary values (for example, a flow execution key) in the request path. In other words, you can also use the request parameters discussed in the previous section with this argument handler.

For instance, the following anchor is equivalent to the one shown in the "Launching Flows" section; it launches the `enterPayment-flow`:

```
<a href="<c:url value="/flows/enterPayment-flow.html"/>">
  <fmt:message key="enterPayment"/>
</a>
```

To make this work, a slightly modified request mapping is required. The front controller of the web MVC framework (for example, the `DispatcherServlet` in the case of Spring Web MVC) is still mapped to all `*.html` requests. The Spring Web Flow controller, on the other hand, is now mapped to all requests matching with `/flows/*`, instead of just `/flows.html`, which would obviously not work. Here is the REST-style URL generated by the preceding code:

```
/springbank/flows/enterPayment-flow.html
```

To participate in an ongoing flow execution, the flow execution key can be appended to the end of the resource path, using `/k/` as a delimiter:

```
/springbank/flows/k/_c50..._kE3...html
```

The preceding URL would trigger a flow execution refresh because it contains a flow execution key but no event ID. The `_eventId` value still has to be submitted as a normal request parameter, for instance:

```
/springbank/flows/k/_c50..._kE3...html?_eventId=next
```

You can, of course, also use HTML forms in combination with REST-style URLs. In that case, there is no need for hidden form fields that submit the flow ID or flow execution key, since those values will already be embedded in the action of the form:

```
<form action="<c:url value="/flows/k/${flowExecutionKey} "/>">
  <input type="hidden" name="_eventId" value="next"/>
  <input type="hidden" name="debitAccount" value="${account.number} "/>
  <input type="submit" value="${account.number} "/>
</form>
```

■ **Note** Only the flow ID and flow execution key are embedded in the path of the resource when using REST-style URLs. The event ID, along with all application-specific values, still needs to be submitted using normal request parameters.

Host Framework Integrations

The general pattern followed when integrating Spring Web Flow into a hosting framework was discussed in the "Flow Executor Integration" section. Integrations with request-based frameworks such as Spring Web MVC or Struts follow this pattern very closely. Integrating with a component-based framework such as JSF is more complex, because component-based frameworks typically enforce a strict request life cycle and provide multiple extension points.

Keep in mind that the following sections explain how to integrate Spring Web Flow into a number of web MVC frameworks. They do not try to cover web application development using those frameworks in any detail. For more information, refer to the relevant documentation of each framework.

INTEGRATE OR CONVERT?

One question that is frequently asked on the Spring Web Flow forums is this: should you integrate Spring Web Flow into an application or convert the entire application to use Spring Web Flow? Unfortunately, there is no easy answer.

In essence, Spring Web Flow was designed as a controller component for web MVC frameworks that targets *controlled navigation* use cases. It complements simple controllers handling *free browsing* requirements. In this line of thinking, it is natural to combine both simple controller implementations, as offered by the hosting framework, and Spring Web Flow in a single application. The Spring Bank sample application illustrates this approach. It uses Spring Web Flow for those parts of the application that have a real flow, like entering a new payment. Simple Spring Web MVC controllers are used for other parts of the application, however, such as logging out or displaying the account balances of a user. The advantage is this approach is that it allows you to use the right tool for the job in each situation, taking maximum advantage of each tool's strengths. Using multiple development techniques in a single application does introduce extra complexity however. Developers have to learn how to work with each technique and think about what option to use in each case.

On the other hand, you can decide to use Spring Web Flow for all use cases in your application. Of course, you will still need a hosting framework, but application developers will mostly use Spring Web Flow to develop controller functionality. This approach has the advantage of allowing you to set up a single, consistent development style for the entire application. The fact that Spring Web Flow's development model scales well from simple uses cases to very complex requirements makes this possible. The downside here is that, for the simple use cases, you will have to deal with some extra complexity, for instance, the configuration and runtime overhead imposed by Spring Web Flow's state management.

In the end, whether to integrate or convert is subjective, determined by your own preferences and the culture and preferences of the application's development team.

Spring Web MVC

The example applications developed in previous chapters all used the Spring Web MVC framework as the hosting framework. Integrating Spring Web Flow with Spring Web MVC is as simple as defining a Spring Web Flow `FlowController`:

```
<bean id="flowController"
  class="org.springframework.webflow.executor.mvc.FlowController">
  <property name="flowExecutor" ref="flowExecutor"/>
</bean>
```

The `FlowController` is a Spring Web MVC `Controller` implementation (it subclasses `AbstractController`) that wraps incoming requests in a `ServletExternalContext`, extracts the relevant flow executor arguments, and delegates to the configured `FlowExecutor`.

The `ServletExternalContext` class is an `ExternalContext` implementation that wraps a `ServletContext`, an `HttpServletRequest`, and an `HttpServletResponse`.

In addition to the mandatory `flowExecutor` property, the `FlowController` also allows you to configure the `FlowExecutorArgumentHandler` it uses. By default, this will be the `RequestParameterFlowExecutorArgumentHandler`, but you can also plug in the `RequestPathFlowExecutorArgumentHandler` or a custom argument handler:

```
<bean id="flowController"
  class="org.springframework.webflow.executor.mvc.FlowController">
  <property name="flowExecutor" ref="flowExecutor"/>
  <property name="argumentHandler">
    <bean class="org.springframework.webflow.executor.support. \
      RequestPathFlowExecutorArgumentHandler"/>
  </property>
</bean>
```

■**Tip** A single flow controller can manage the execution of all flows in your application. This minimizes configuration overhead. Having multiple flow controllers in a single application is also possible, and can be useful in a number of situations such as when you want to spread your flow definitions over multiple flow definition registries, when you do not want to expose all flows of your application under the same URL (for instance, to be able to do URL-based security checks), and when you want to use a differently configured flow executor for particular flows. Using the default continuation repository for some flows, but the single key repository for others, could be an example.

The names of the views selected by Spring Web Flow (determined by the `view` attribute of a view or end state), will be mapped to actual view templates using Spring Web MVC's `ViewResolver` mechanism.

Developing views for a web flow in the context of a Spring Web MVC application was explained in the "Spring Web Flow View Development" section. You can use any view technology supported by Spring Web MVC. When using JSP pages, you can use Spring's binding tag library, as well as the form tag library, for instance:

```
<form:form action="flows.html" commandName="payment">

  ...
  <td colspan="2">
    <fmt:message key="amount"/>:
    <form:input path="amount" /> &euro;
    <form:errors cssClass="error" path="amount"/>
  </td>

  ...

  <input type="hidden" name="_flowExecutionKey"
    value="${flowExecutionKey} "/>
  <input type="submit" name="_eventId_next"
    value="<fmt:message key="next"/>"/>
  <input type="submit" name="_eventId_cancel"
    value="<fmt:message key="cancel"/>"/>
</form:form>
```

Spring Portlet MVC

The Spring Portlet MVC framework closely follows the design of the Spring Web MVC framework but applies it to a Portlet environment (Java Community Process 2003). Integrating Spring Web Flow with Spring Portlet MVC is similar to integrating it with Spring Web MVC. Integration simply boils down to defining a `PortletFlowController` in the application context of the `DispatcherPortlet`:

```
<bean id="flowController"
  class="org.springframework.webflow.executor.mvc.PortletFlowController">
  <property name="flowExecutor" ref="flowExecutor"/>
</bean>

<flow:executor id="flowExecutor" registry-ref="flowRegistry">
  <flow:execution-attributes>
    <flow:alwaysRedirectOnPause value="false"/>
  </flow:execution-attributes>
</flow:executor>
```

The `PortletFlowController` wraps `PortletContext`, `PortletRequest`, and `PortletResponse` inside `PortletExternalContext`. Just like the `FlowController`, the `PortletFlowController` also uses a `FlowExecutorArgumentHandler` to extract `FlowExecutor` arguments. The `FlowExecutorArgumentHandler` and `FlowExecutor` used can be configured with the `argumentHandler` and `flowExecutor` properties respectively.

When using Spring Web Flow in a Portlet environment, you should be aware of a few technical constraints caused by peculiarities in the Portlet specification:

- The Portlet specification already identifies separate render and action requests. This makes use of the POST-REDIRECT-GET idiom and "always redirect on pause" impossible in a Portlet environment. Notice how the flow executor configuration in the preceding example disables "always redirect on pause".
- The `PortletFlowController` uses the `PortletSession` to pass information from the Portlet action request to the render request. This means that Spring Web Flow Portlet applications will always require a session, even when using the client continuation repository.
- The `PortletFlowController` launches new flow executions in the Portlet render request. This has two implications. First, refreshing the first page of a flow will always cause a new flow execution to be launched. Second, the first view rendered by the flow cannot be a redirect, because a Portlet render request cannot issue a redirect.

Views selected by Spring Web Flow are mapped to view implementations using the installed `ViewResolver`, just as in the case of the Spring Web MVC integration.

Developing views that participate in a web flow running in a Portlet environment is not much different from developing a normal Portlet page. Just follow the general guidelines described in the "Spring Web Flow View Development" section, and use the `<portlet:actionURL>` custom tag to build request URLs:

```
<a href="
  <portlet:actionURL>
    <portlet:param name="_flowExecutionKey"
      value="<%=request.getAttribute("flowExecutionKey") %>" />
    <portlet:param name="_eventId" value="next" />
    <portlet:param name="debitAccount"
      value="<%=((Account)pageContext.getAttribute("account")). \
        getNumber().toString() %>"/>
  </portlet:actionURL>">
  ${account.number}
</a>
```

Struts

Spring Web Flow also integrates with the classic Apache Struts web MVC framework. In Struts, controller components are represented as `Action` classes.[1] Spring Web Flow integrates into Struts using a `FlowAction`, which is defined as any other Struts action in `struts-config.xml`:

```
<action-mappings>

  ...

  <action path="/flowAction" name="actionForm" scope="request"
    type="org.springframework.webflow.executor.struts.FlowAction"/>

  ...

</action-mappings>
```

If you're using the preceding configuration, all `/flowAction.do` requests will be handled by Spring Web Flow.

The `FlowAction` acts as a flow executor front-end. It wraps the incoming request and response in a `StrutsExternalContext`, extracts any relevant flow executor arguments, and invokes the flow executor. The `FlowAction` will look for beans named `flowExecutor` and `argumentHandler` in the root web application context to determine the `FlowExecutor` and `FlowExecutorArgumentHandler` to use. To set up a root web application context, you'll have to add Spring's `ContextLoaderListener` to the `web.xml` deployment descriptor:

```
<web-app ...>

  ...

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/webflow-config.xml</param-value>
  </context-param>
```

1. Don't confuse a Struts `org.apache.struts.action.Action` with a Spring Web Flow `org.springframework.webflow.execution.Action`. The two concepts are unrelated.

```
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  ...

</web-app>
```

Since the referenced `webflow-config.xml` file is a normal Spring application context XML file, you can use Spring Web Flow's configuration schema to define the flow executor, argument handler, flow definition registry, and any other required beans. Keep in mind, however, that the bean name of the `FlowExecutor` bean needs to be `flowExecutor`, and `argumentHandler` needs to be the bean name for the `FlowExecutorArgumentHandler` bean.

Configuring the flow executor and argument handler as described previously has the disadvantage that all `FlowActions` in your application use the same flow executor and argument handler. As an alternative `FlowAction` configuration mechanism, you can also use Spring's `DelegatingActionProxy` (or `DelegatingRequestProcessor`). This will allow you to configure Struts actions as normal Spring beans in an application context, taking advantage of Spring's dependency injection techniques to configure your `FlowAction` implementation or implementations. Consult the Spring reference documentation for more details on its Struts support (Johnson et al 2003).

Tip You can use a single `FlowAction` to drive all flow executions in your Struts application. This will drastically cut down on the number of action definitions you have to add to the `struts-config.xml` file.

To map the views selected by Spring Web Flow to actual JSP pages, you need to add the necessary forwards to the `struts-config.xml` file. If you're using a single `FlowAction` to execute multiple flows, you'll typically want to use global forwards:

```
<global-forwards>
  <forward name="selectDebitAccount"
    path="/WEB-INF/jsp/selectDebitAccount.jsp"/>
  <forward name="enterPaymentInfo"
    path="/WEB-INF/jsp/enterPaymentInfo.jsp"/>

  ...

</global-forwards>
```

The general view development guidelines covered in the “Spring Web Flow View Development” section also apply when using Spring Web Flow in combination with Struts. Displaying data binding and validation messages deserves some special attention however.

Data Binding and Validation

Struts has its own data binding and validation mechanism based on `ActionForm` objects. Those `ActionForm` objects are stored in either the `HttpServletRequest` or the `HttpSession`, in other words, outside of a Spring Web Flow flow execution. You can use classic Struts `ActionForm` objects in combination with Spring Web Flow, which leaves data binding and validation to Struts and lets the flow access the data in the `ActionForm` object in the request or session.

An alternative is to not use Struts `ActionForm` objects at all, and instead use Spring Web Flow’s `FormAction`, as discussed in the “The Form Action” section of Chapter 5. The downside of this approach is that you will no longer be able to use the traditional Struts tag libraries to display form-backing object information and validation errors. Instead, you’ll have to use Spring’s binding and form tag libraries.

There is a third approach based on Spring’s `SpringBindingActionForm`, for which the `FlowAction` has special support. Using `SpringBindingActionForm` allows you to leverage the `FormAction` and still use the traditional Struts tag libraries. All you need to do is use `SpringBindingActionForm` as the form bean type in `struts-config.xml` and reference that form bean from your `FlowAction` definition:

```
<form-bean name="actionForm"
  type="org.springframework.web.struts.SpringBindingActionForm"/>
```

The `SpringBindingActionForm` will automatically pick up on the form-backing object and `Errors` instance managed by Spring Web Flow’s `FormAction`. You can then use that `FormAction` to do data binding and validation from inside your flow and display the results in the JSP page with the help of the traditional Struts tag libraries:

```
<html:form action="flowAction" method="post">

  ...

  <td colspan="2">
   Amount:
   <html:text property="amount" errorStyleClass="error"/>
   &euro;
  </td>

  ...
  <input type="hidden" name="_flowExecutionKey"
    value="${flowExecutionKey} "/>
  <html:button property="_eventId_next" value="Next"/>
  <html:button property="_eventId_cancel" value="Cancel"/>
</html:form>
```

JavaServer Faces

The integration of Spring Web Flow into JavaServer Faces (JSF) is elegant and powerful. Under the covers, it is quite a bit more complex than the integrations with other web MVC frameworks, but luckily, this does not impact you as an application developer. A primary goal of Spring Web Flow's integration with JSF is making it feel natural for JSF developers.

When integrated into JSF, Spring Web Flow fulfills two important roles:

- It handles navigation between JSF views, allowing you to use flow definitions to define the allowed navigational paths.

- It resolves variables referenced by JSF expressions. This will allow JSF components to access model information stored in one of Spring Web Flow's flow execution scopes.

Both responsibilities naturally complement JSF's component model, making the combination of JSF and Spring Web Flow an attractive offering. Plain JSF views can participate in a flow execution and can transparently access information managed by that flow execution. Furthermore, other popular JSF view technologies such as Facelets (`https://facelets.dev.java.net`), ICEfaces (`http://www.icefaces.org`), and Apache MyFaces Trinidad (`http://myfaces.apache.org/trinidad`) are not impacted by all of this.

Configuration

Configuring JSF to use Spring Web Flow requires some simple definitions in JSF's `faces-config.xml` configuration file:

```
<faces-config>
  <application>
    <navigation-handler>
      org.springframework.webflow.executor.jsf.FlowNavigationHandler
    </navigation-handler>
    <variable-resolver>
      org.springframework.webflow.executor.jsf.DelegatingFlowVariableResolver
```

```
 8      </variable-resolver>
 9    </application>
10
11    <lifecycle>
12      <phase-listener>
13        org.springframework.webflow.executor.jsf.FlowPhaseListener
14      </phase-listener>
15    </lifecycle>
16  </faces-config>
```

Here are the key points to note about the preceding listing:

- *Line 3*: The `FlowNavigationHandler` will signal a JSF action outcome, such as a command button click, as an event in the active flow execution.

■**Tip** If a request does not participate in an ongoing flow execution, the `FlowNavigationHandler` will simply delegate to the standard JSF `NavigationHandler`, which reads navigation rules from `faces-config.xml`. This allows you to use Spring Web Flow for some use cases in your application and standard JSF for other situations.

- *Line 5*: The `DelegatingFlowVariableResolver` will try to resolve normal JSF binding expressions, such as `#{payment.amount}`, to objects managed in one of Spring Web Flow's flow execution scopes. The flash, flow, and conversation scopes will be scanned, in that order, until a match is found. If no matching object can be located, the `DelegatingFlowVariableResolver` will delegate to the next variable resolver in JSF's variable resolver chain.

■**Note** Notice that the `DelegatingFlowVariableResolver` does not resolve request scope variables. As a consequence, you cannot reference objects in request scope from JSF components using JSF's expression language. Binding components onto request-scoped objects is problematic in a JSF environment, because those objects are longer available when JSF restores the component view state on a subsequent postback request.

Recall the best practice of putting large objects in the request scope and reloading them every time using a view state render action. Since the request scope is not available using Spring Web Flow's JSF integration, you can no longer strictly follow this best practice. Instead, consider loading large objects using a view state entry action and storing them in the flash scope.

- *Line 12*: Spring Web Flow's `FlowPhaseListener` manages flow executions in a JSF environment. As a flow executor front-end, it launches, resumes, and refreshes flow executions as required. The `FlowPhaseListener` wraps the incoming JSF request (a `FacesContext`) in a `JsfExternalContext` and makes the flow execution available to other JSF artifacts that might need access to it, such as variable resolvers.

ACCESSING THE FLOW EXECUTION FROM PLAIN JSF ARTIFACTS

You can access the flow execution from inside your own JSF artifacts, for instance an `ActionListener`, as follows:

```
FacesContext context = FacesContext.getCurrentInstance();
FlowExecution flowExecution =
  FlowExecutionHolderUtils.getRequiredCurrentFlowExecution(context);
```

This will obtain the flow execution from a thread locally managed by the `FlowPhaseListener`.

As you can see from the preceding configuration, Spring Web Flow's JSF integration does not directly use a `FlowExecutor`. To make configuration consistent with the other environments, the `FlowPhaseListener` and `FlowNavigationHandler` automatically pick up any relevant configuration from a `flowExecutor` bean defined in the root web application context. Consequently, you have to set up a root web application context, typically using Spring's `ContextLoaderListener` in the `web.xml` deployment descriptor of the web application:

```
<web-app ...>

  ...

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/webflow-config.xml</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  ...

</web-app>
```

In the preceding example, the root web application context will simply contain Spring Web Flow configuration. Of course, it can also contain other supporting services used by the application. You can define the flow executor as usual, along with a flow definition registry, and possibly a conversation manager. Just make sure to use the `flowExecutor` name for the `FlowExecutor` bean:

```
<flow:executor id="flowExecutor" registry-ref="flowRegistry"/>

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

■**Caution** Spring Web Flow 1 does not allow you to have multiple flow executors, each with a different configuration, in a single JSF application. You might be able to work around this limitation, for instance, using multiple FacesServlets and Spring's delegating JSF proxies (Johnson et al 2003), but this workaround is not supported out of the box.

System Cleanup

The JSF specification is not sufficiently strict to allow the `FlowPhaseListener` to clean up the Spring Web Flow artifacts related to a request in all possible situations. For instance, JSF will never call back into the `FlowPhaseListener` if fatal errors occur during request processing. To work around this problem and make sure everything is cleaned up properly for every request, Spring Web Flow includes a `FlowSystemCleanupFilter`, which you can configure in `web.xml`:

```
<web-app ...>

  ...

  <filter>
    <filter-name>cleanupFilter</filter-name>
    <filter-class>
      org.springframework.webflow.executor.jsf.FlowSystemCleanupFilter
    </filter-class>
  </filter>
  <filter-mapping>
    <filter-name>cleanupFilter</filter-name>
    <url-pattern>*.faces</url-pattern>
  </filter-mapping>

  ...

</web-app>
```

Typically, you'll want to map the FlowSystemCleanupFilter to the same URL pattern as the JSF FacesServlet (*.faces in the preceding example).

JSF AND SPRING WEB FLOW BEFORE VERSION 1.0.2

The DelegatingFlowVariableResolver was only introduced in Spring Web Flow version 1.0.2. Earlier versions used primitive FlowVariableResolver and FlowPropertyResolver implementations that could only resolve variables in flow scope. Although this code is still available, using the DelegatingFlowVariableResolver instead is recommended. Consult the Spring Web Flow API and reference documentation for more information on these classes.

Launching flows

To allow standard JSF components to launch new flow executions, Spring Web Flow's JSF integration recognizes JSF action outcomes that adhere to a special format. An action outcome that has the flowId: prefix will launch the identified flow, for instance:

```
<h:commandLink value="Enter Payment" action="flowId:enterPayment-flow"/>
```

The preceding command link launches the enterPayment-flow. Alternatively, you can also use a simple URL targeted at the JSF FacesServlet that contains the _flowId request parameter and no _flowExecutionKey request parameter:

```
<a href="controller.faces?_flowId=enterPayment-flow">Enter Payment</a>
```

This is in line with the way flows are launched in the other environments Spring Web Flow integrates with.

Developing Flows in a JSF Environment

Let's take a look at how to develop flows when using Spring Web Flow combined with JSF.

Flow Definitions

JSF is a component-based web MVC framework. JSF components elegantly handle data binding and validation logic. With the help of the DelegatingFlowVariableResolver, JSF components can directly bind onto objects stored in one of the flow execution scopes. As a result, data binding and validation will no longer have to be integrated into the flow definition using the FormAction. This leads to flow definitions that are generally simpler and more focused on expressing navigational rules. Here is an extract that shows what a section of the enterPayment-flow would look like in a JSF environment:

```
<view-state id="showSelectDebitAccount" view="/selectDebitAccount.jsp">
  <entry-actions>
    <bean-action bean="accountRepository" method="getAccounts">
      <method-arguments>
        <argument expression="externalContext.sessionMap.user.clientId"/>
      </method-arguments>
      <method-result name="accounts" scope="flash"/>
    </bean-action>
  </entry-actions>
  <transition on="next" to="showEnterPaymentInfo"/>
</view-state>

<view-state id="showEnterPaymentInfo" view="/enterPaymentInfo.jsp">
  <transition on="next" to="showConfirmPayment"/>
  <transition on="selectBeneficiary" to="launchBeneficiariesFlow"/>
</view-state>
```

As you can see, there are no references to the FormAction. Also notice that the view names used in the view states above use the standard JSF view identifier format, using a leading forward slash and ending with a suffix. The suffix is .jsp in this case, but it could be something else; for instance, it's .xhtml when using Facelets. JSF will directly map these view names to a corresponding view implementation. Also notice how the accounts list resulting from the bean action is explicitly stored in the flash scope. This is necessary, since JSF views cannot reference request-scoped objects, as mentioned previously.

View Development

JSF views participating in a web flow are plain JSF views, with no Spring Web Flow–specific elements. This is true regardless of the view technology used, for example, JSP or Facelets. Here is an extract from a JSF view using JSP technology:

```
<f:view>

  ...

  <h:form id="paymentInfoForm">

    ...

    <td colspan="2">
      Amount:
      <h:inputText id="amount" value="#{payment.amount} " required="true">
        <f:validateDoubleRange minimum="0.01"/>
      </h:inputText>
      &euro;
    </td>

    ...

    <h:commandButton type="submit" value="Next" action="next"/>
    <h:commandButton type="submit" value="Cancel" action="cancel"/>
  </h:form>

  ...

</f:view>
```

The flow execution key is automatically tracked in the JSF view root, so there is no need to manually submit a `_flowExecutionKey` request parameter with every request. The action outcomes specified by the command buttons in the preceding example will automatically signal the corresponding event in the flow execution. Also notice how the JSF expression `#{payment.amount}` transparently accesses the `payment` object in the conversation scope.

Facelets Facelets (`https://facelets.dev.java.net`) is an alternative view definition technology for JSF. It was explicitly designed for use with JSF and solves many of the problems related to using JSP to define JSF views. Since Facelets is very popular among JSF developers, quickly showing its usage in combination with Spring Web Flow seems useful.

Configuring a JSF application to use Facelets is easy. You simply need to add the required JAR files to the application and add a context parameter to the `web.xml` deployment descriptor to instruct JSF to use Facelets XHTML files as view definitions:

```
<context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.xhtml</param-value>
</context-param>
```

Next, you have to configure JSF to use the Facelets view handler by adding the following to your `faces-config.xml`:

```
<application>

  ...

  <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
</application>
```

This view handler complements the Spring Web Flow–specific configuration in `faces-config.xml`. The application is now set up to use Facelets. As far as Spring Web Flow is concerned, the only impact of using Facelets is the slightly different view IDs specified in the flow definition:

```
<view-state id="showEnterPaymentInfo" view="/enterPaymentInfo.xhtml">
  <transition on="next" to="showConfirmPayment"/>
  <transition on="selectBeneficiary" to="launchBeneficiariesFlow"/>
</view-state>
```

View names now end with the `.xhtml` suffix. As was the case with JSP view definitions, the Facelets view definitions themselves contain no Spring Web Flow–specific details.

■ **Note** JSF puts the view developer at a slightly higher level of abstraction. Instead of directly creating requests that submit a flow execution key and an event, you can use JSF actions to signal events while the flow execution key is automatically tracked. Internally, JSF constructs the actual requests. Since controlling the `FlowExecutorArgumentHandler` does not really make sense in this context, the JSF integration doesn't provide a direct way to configure the argument handler used.

Deep Integration Between JSF and Spring Web Flow

The previous examples illustrate how Spring Web Flow's JSF integration allows you to use value bindings to bind JSF components to model objects managed inside a flow execution. For instance, the following JSF data table definition directly uses the accounts list of Account objects available in the flash scope:

```
<h:dataTable value="#{accounts}" var="account" border="1">

  ...

  <h:column>
    <h:commandLink action="next">
      <f:param name="debitAccount" value="#{account.number}"/>
      <h:outputText value="#{account.number}"/>
    </h:commandLink>
  </h:column>

  ...

</h:dataTable>
```

The JSF data table component will automatically convert the List<Account> to a JSF ListDataModel. In this case, the flow is unaware of JSF's rich component model. Consequently, the command link used to select one of the listed accounts needs to explicitly submit the relevant account number as a request parameter.

This style of JSF usage essentially uses JSF as a powerful tag library, similar to the way Displaytag is used when using Spring Web MVC. Some people prefer this style, and it is certainly a simple and elegant approach. The downside is that you can't leverage JSF's powerful component model. To be able to do that, you need to take a slightly different approach.

In classic JSF development, it is typical to have a backing bean manage all data and behavior required for a particular page. You can mimic this way of working using flow variables and evaluate actions. Recall the way evaluate actions can be used to directly invoke methods on objects stored inside the flow execution, typically flow variables; this was explained in the "Evaluate Actions" section of Chapter 5. Using a flow variable, you can set up a flow-backing bean in one of the flow execution scopes:

```
<var name="backingBean" scope="conversation"/>
```

This flow-backing bean will manage all data and behavior required by the flow, just like a traditional JSF-backing bean. The flow can invoke functionality provided by the backing bean using evaluate actions:

```
<view-state id="showSelectDebitAccount" view="/selectDebitAccount.jsp">
  <entry-actions>
    <evaluate-action
      expression="conversationScope.backingBean.loadAccounts()"/>
  </entry-actions>
  <transition on="next" to="showEnterPaymentInfo">
    <evaluate-action
      expression="conversationScope.backingBean.setDebitAccount()"/>
  </transition>
</view-state>
```

Notice how the first evaluate action in this listing does not directly expose the list of loaded accounts to the flow. It is the responsibility of the backing bean to do that:

```
public class EnterPaymentFlowBackingBean implements Serializable {

  private transient AccountRepository accountRepository;
  private Long clientId;
  private Payment payment = new Payment();
  private ListDataModel accounts;

  public void setAccountRepository(AccountRepository accountRepository) {
    this.accountRepository = accountRepository;
  }

  public void setClientId(Long clientId) {
    this.clientId = clientId;
  }

  public Payment getPayment() {
    return payment;
  }

  public ListDataModel getAccounts() {
    return accounts;
  }

  public void setAccounts(ListDataModel accounts) {
    this.account = accounts;
  }

  public void loadAccounts() {
    accounts = new ListDataModel(accountRepository.getAccounts(clientId));
  }

  public void setDebitAccount() {
    payment.setDebitAccount((Account)accounts.getRowData());
34    }
35
36    ...
37
38  }
```

There are several interesting things to highlight in the preceding code fragment:

- *Line 1*: Because the flow-backing bean is stored inside the flow execution, it needs to be serializable (especially when it would be stored in the flow scope).

- *Line 3*: The backing bean obviously needs access to the `AccountRepository` to be able to load the list of accounts. Since the backing bean is defined as a normal Spring bean, we can simply inject the `AccountRepository`:

```
<bean id="backingBean" scope="prototype"
  class="com.ervacon.springbank.web.EnterPaymentFlowBackingBean">
  <property name="accountRepository" ref="accountRepository"/>
</bean>
```

There is one problem however: the backing bean needs to be serializable! To avoid the `accountRepository` from being serialized along with the rest of the backing bean, it is marked as *transient*. This introduces another problem: when the backing bean is restored from its serialized state, the `accountRepository` member will be null. The solution to this problem is to *rewire* all objects in the flow execution when the flow execution is restored. Spring Web Flow provides a specialized JSF `AutowiringPhaseListener` to do this. You need to add this phase listener to `faces-config.xml`, along with the `FlowPhaseListener`:

```
<lifecycle>
  <phase-listener>
    org.springframework.webflow.executor.jsf.FlowPhaseListener
  </phase-listener>
  <phase-listener>
    org.springframework.webflow.executor.jsf.AutowiringPhaseListener
  </phase-listener>
</lifecycle>
```

The `AutowiringPhaseListener` will automatically rewire all objects found in any of the flow execution scopes *by name*. In the case of the `enterPayment-flow` backing bean, it will reinject the `AccountRepository` bean.

- *Line 4*: The `clientId` of the currently logged in user will be stored in the HTTP session, as you will see in Chapter 9. Using Spring 2's scoped bean feature, which was already discussed in the "Accessing Scoped Beans" section, we can make the client ID of the current user available to the backing action.

- *Line 6*: The backing bean directly holds on to a `ListDataModel` instance. The standard JSF `ListDataModel` is not serializable. To work around that problem, Spring Web Flow provides serializable versions of the JSF model classes in the `org.springframework.webflow.executor.jsf.model` package.

The JSF data table used to display the list of debit accounts now becomes quite a bit simpler:

```
<h:dataTable value="#{backingBean.accounts}" var="account" border="1">

  ...

  <h:column>
    <h:commandLink action="next">
      <h:outputText value="#{account.number}"/>
    </h:commandLink>
  </h:column>

  ...

</h:dataTable>
```

Note how it is no longer required to submit the account number of the selected account as a request parameter. The JSF data table will automatically track the selected account in the `ListDataModel`, making it available using `getRowData()`. The `EnterPaymentFlowBackingBean` listing shown earlier illustrated this.

ACCESSING JSF COMPONENTS FROM INSIDE A FLOW

You can also use *component bindings* to make actual JSF component instances available to the backing bean, for instance:

```
<h:panelGroup binding="#{backingBean.warningPanel}">
  ...
</h:panelGroup>
```

This allows you to directly manipulate the components from inside your backing bean.

Summary

The flow executor front-ends discussed in this chapter integrate Spring Web Flow into several popular web MVC frameworks.

The chapter started off explaining the general approach to integrating Spring Web Flow into a hosting framework and introduced the concept of a `FlowExecutorArgumentHandler` along the way. General guidelines related to Spring Web Flow view development were also covered, detailing the model data exposed to the view and how to build requests that launch a new flow execution, signal an event, or refresh a flow execution.

Integrating Spring Web Flow into request-based web MVC frameworks is straightforward. There is a `FlowController` for Spring Web MVC, a `PortletFlowController` for Spring Portlet MVC, and a `FlowAction` for Struts. All of these integrations simply delegate to a `FlowExecutor` implementation.

Integrating with JSF, a component-based web MVC framework, requires a slightly different approach. In a JSF environment, Spring Web Flow provides advanced navigation-handling capabilities and transparent resolution of variables that live in a flow execution. The end result is a development experience that fits very naturally with what JSF developers are used to. JSF views participating in a web flow contain no Spring Web Flow–specific elements. Even the flow execution key is tracked automatically.

You now have all the details of using Spring Web Flow in a runtime environment. You know how to define a flow, how to set up a flow executor to manage executions of that flow, and how to integrate that flow executor with your hosting framework of choice. The next chapter will explain how to use your flows in a test environment.