Spring Web Flow 2 Tutorial

by

Ivan A Krizsan

Version: June 2, 2010

Table of Contents

Purpose

This document contains some introductory examples and advice concerning using Spring Web Flow 2 to create sequences of web pages, called flows, in web applications. The level is introductory.

Licensing

Disclaimers

Introduction

Spring Web Flow 2 allows for creation of flows in web applications. A flow is composed of a set of web pages used in a certain order associated with some activity. For instance, the web pages required to search for a hotel, select and book a hotel and, finally, receive a confirmation of the booking.
Spring Web Flow can be used with different kinds of web frameworks or technologies; for instance JSP, Spring MVC, Java ServerFaces. Spring Web Flow is less well suitable for web applications that does not have flows that are known in advance.
This document contains a basic tutorial to Spring Web Flow 2. Only a limited set of features are presented. All the examples use JSP as the presentation technology.

Note that the example programs does not necessary show best habits regarding how to structure a Spring Web Flow application. Focus has been placed on making the examples succinct and to the point regarding the feature(s) they demonstrate.

Prerequisites

Readers are assumed to be familiar with the following subjects:

- Developing Java software with the Eclipse IDE.
- Installing plugins in Eclipse.
  This will not be necessary if you chose to use the customized Eclipse version from SpringSource.
- Developing web applications with Java.
  Basic familiarity is sufficient; for instance with writing JSP pages, configuring a web application's deployment descriptor etc.
- Maven.
  Pom-files will be supplied for all the examples.

An internet connection is required, at least initially, to allow Maven to download the project dependencies.

1. Setting Up the Development Environment

When writing this document, I have been working in the following environment:

- SpringSource Tool Suite 2.3.2
  Alternatively, the standard version of Eclipse with the Spring IDE plugins can be used. In this document, Eclipse and SpringSource Tool Suite will be used interchangeably to denote the IDE used to develop in.
- The m2eclipse Maven Plugin.
  Preinstalled in the SpringSource Tool Suite.
- The Spring IDE Plugin.
  If you use a standard version of Eclipse, you can get the functionality of the SpringSource Tool Suite by installing Spring IDE.
- Apache Tomcat 6

1.1. The IDE

The preferred alternative is to use the SpringSource Tool Suite, a customized version of the Eclipse IDE. It is delivered with a number of plugins preinstalled. All instructions in this document assume that the SpringSource Tool Suite is used.
The SpringSource Tool Suite can be downloaded from the following web page:
http://www.springsource.com/products/springsource-tool-suite-download.

Alternatively, the JavaEE version of Eclipse can be used. The m2eclipse and Spring IDE plugins will have to be installed in this IDE. Eclipse can be downloaded from http://www.eclipse.org.

1.2. Eclipse m2eclipse Maven Plugin

In this document, Maven is used to manage the dependencies. The m2eclipse plugin allows Eclipse projects to use the dependencies declared in a Maven pom.xml file. Installing this plugin is only necessary if you use a standard version of Eclipse, since the SpringSource Tool Suite already contains this plugin.
For more information about the m2eclipse plugin as well as installation instructions, please visit http://m2eclipse.sonatype.org/.

1.3. Eclipse Spring IDE Plugin

If you, for some reason, do not want to install the SpringSource Tool Suite, then you can install the Spring IDE plugin. It can be downloaded from http://www.springsource.com/download/community. Navigate down to the Spring IDE entry and click it to see the available downloads.

1.4. Downloading Spring Web Flow

When writing this tutorial, I used Spring Web Flow 2.0.9, which, along with its documentation, can be downloaded from the same web page as the Spring IDE plugin, that is http://www.springsource.com/download/community. Navigate to the Spring Web Flow entry and click it to see the available downloads.

1.5. Apache Tomcat

In this tutorial I have used Apache Tomcat 6 as the web container. If it is not available as a server in your IDE, the following steps will make it available:

- Download Apache Tomcat 6 from http://tomcat.apache.org.
- Unpack the downloaded archive to a location of your choice.
- Add the Tomcat instance as a server in the IDE.
  Right-click in the Servers view and select New -> Server.
  Specify the location of the Tomcat installation to use and which Java runtime environment to use. For further details, search for “creating a server” in the IDE help (select the Search item in the Help menu).

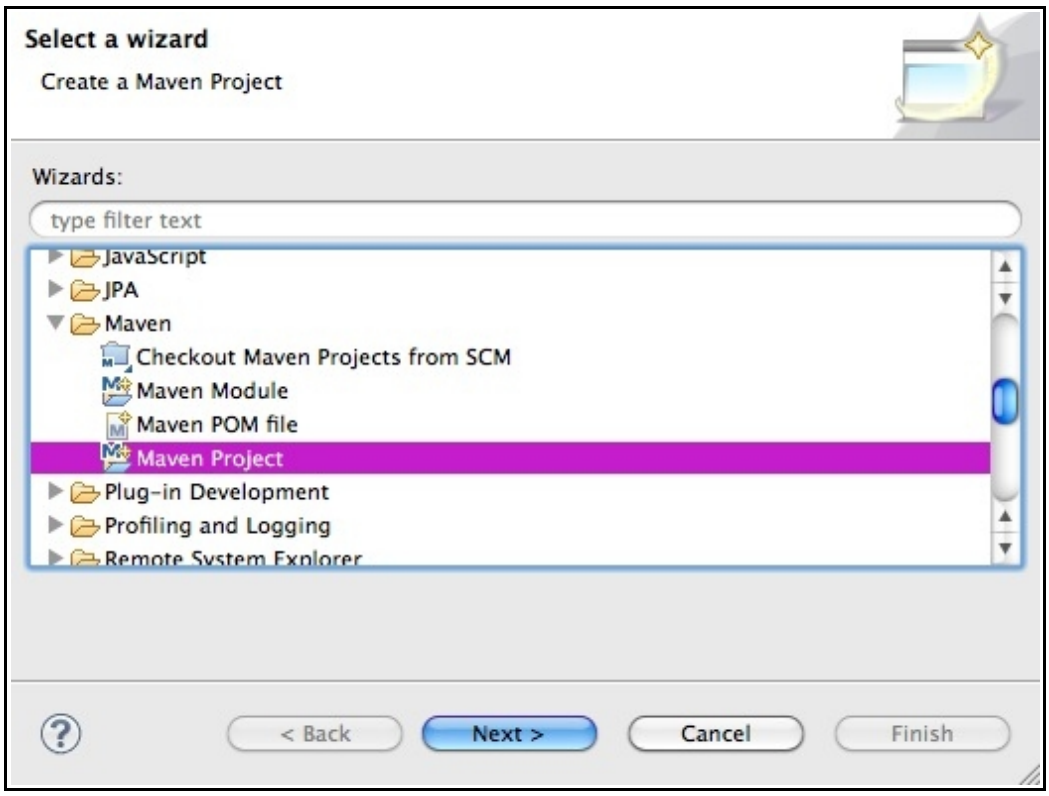
2. Setting Up a Spring Web Flow Project

This chapter describes how to create a new project in the SpringSource Tool Suite IDE and to perform the necessary configurations. The project created here will serve as the base for our further exploration of Spring Web Flow and will be extended step by step as we look at different features.

2.1. Creating the Project

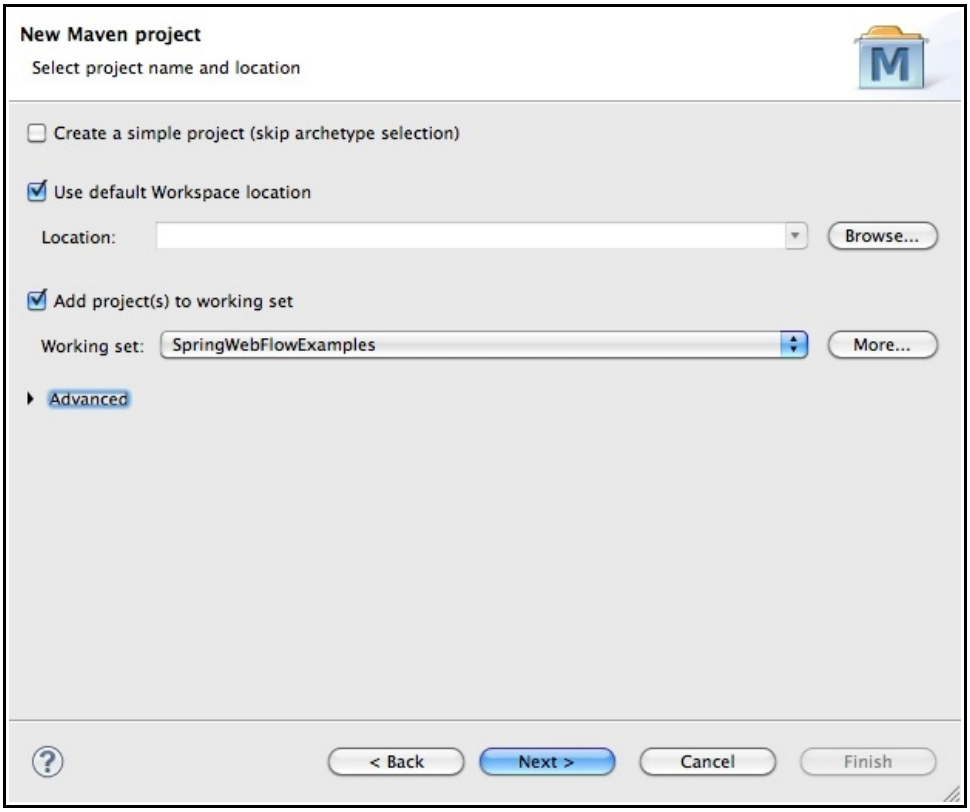
Since we are going to use Maven to manage the dependencies in the tutorial project, we start by creating a new Maven project in the IDE:

- Select the menu File -> New -> Project... in the IDE.
- In the Maven node, select Maven Project:



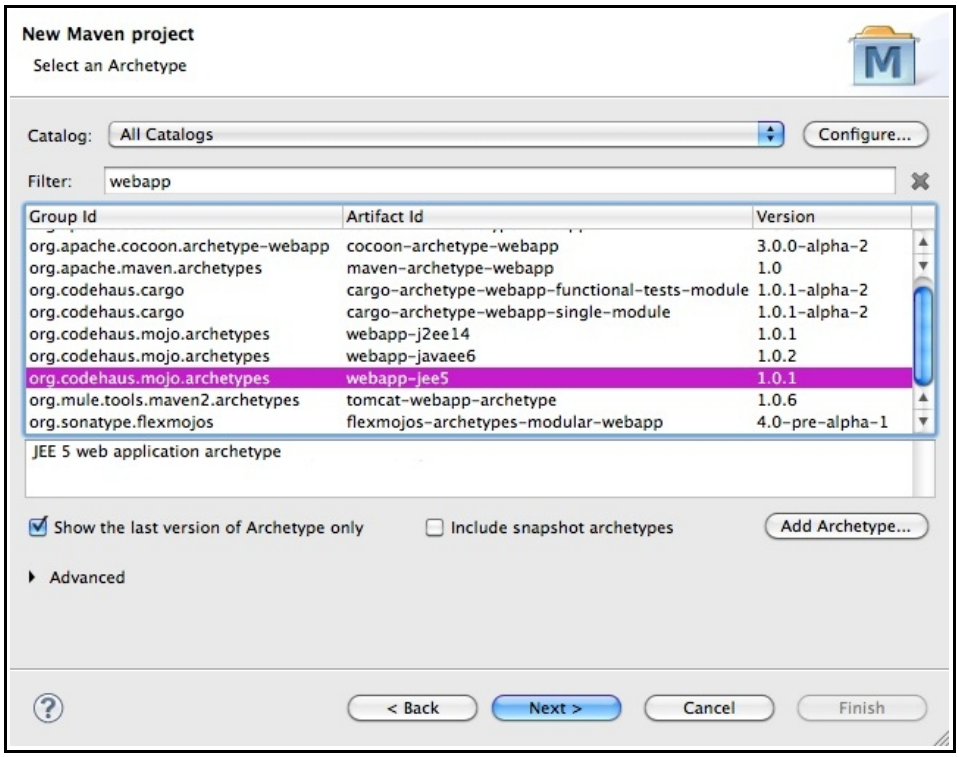
Selecting the Maven Project wizard when creating the new project in the IDE.

- In the next dialog no modifications are needed, since we are going to use a Maven archetype. I have added my project to a working set, but this is not necessary.



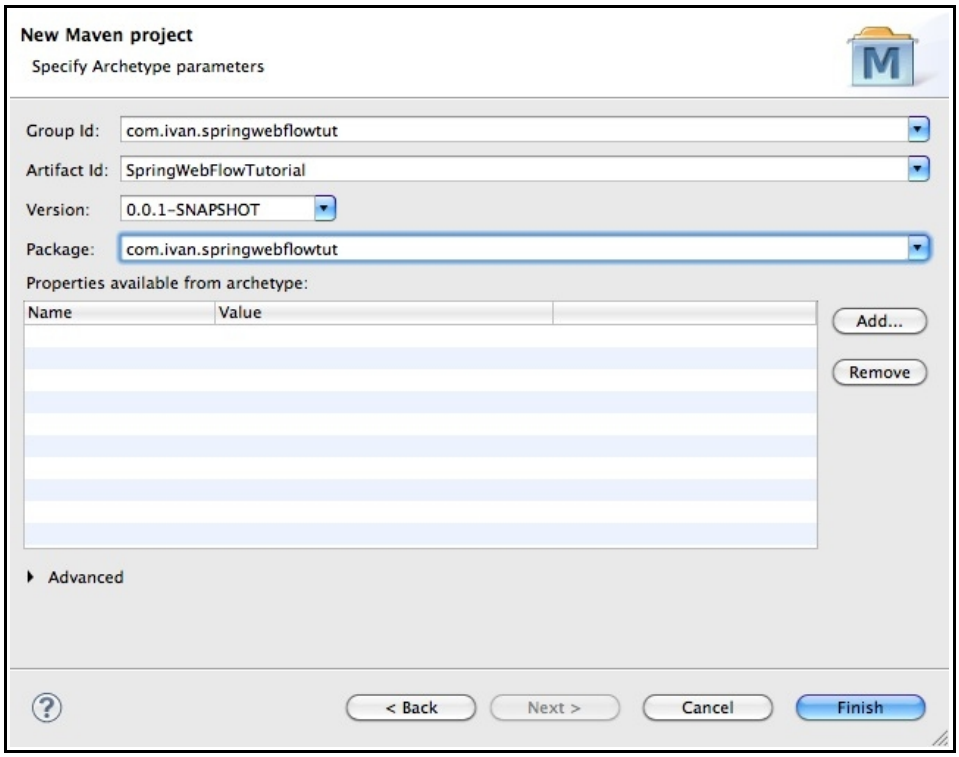
Selecting the Maven project name and location when creating the new project in the IDE.

- Select the Maven archetype with the group id “org.codehaus.mojo.archetypes” and the artifact id “webapp-jee5”. As far as I have been able to determine, this is the archetype that works best when setting up a dynamic web project in Eclipse.



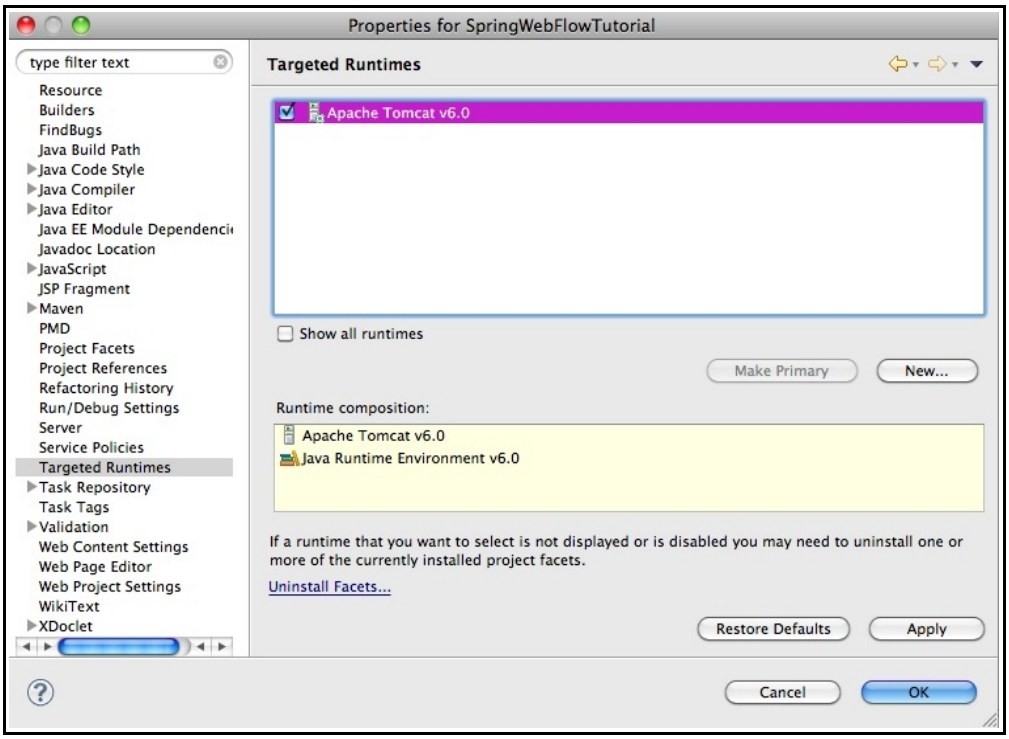
Selecting a Maven archetype when creating the new project in the IDE.

- Specify the Maven archetype parameters according to the figure below:



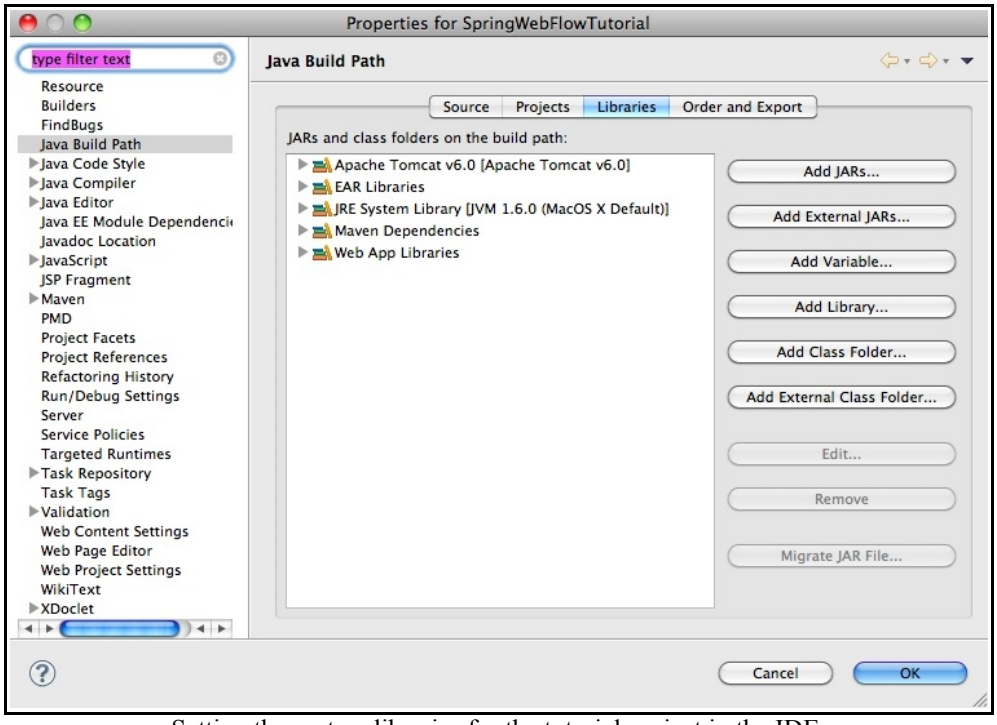
Setting the Maven archetype parameters when creating the new project in the IDE.

- Having clicked the Finish button, the project should now appear in the IDE.
  If it does not, try refreshing the appropriate browser pane or, as a last resort, restart the IDE.
- Set the Targeted Runtime to be Apache Tomcat 6.
  This is accomplished in the project properties according to the figure below.



Setting the targeted runtime for the tutorial project in the IDE.

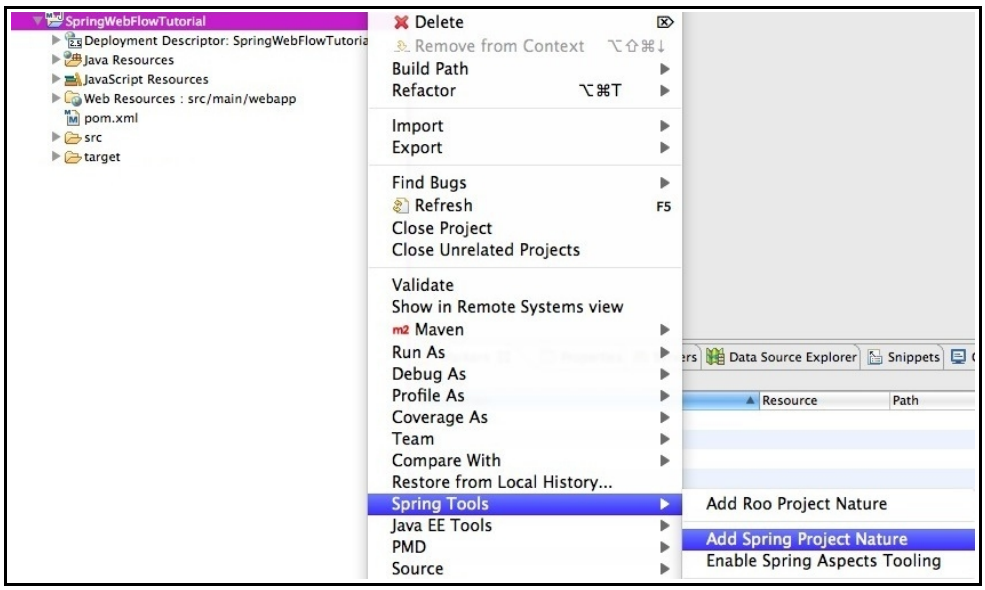
- Optionally, set the system libraries used by the project to the Java 6 libraries.
  This is also done in the project properties, but in the Java Build Path section.



Setting the system libraries for the tutorial project in the IDE.

If you also choose to change the compiler compliance setting, remember that you also need to update the version of the Java project facet to 6.0.

- Finally, the Spring project nature is also added to the project.
  Right-click the project and select Spring Tools -> Add Spring Project Nature.



Adding the Spring Project Nature to the tutorial project in the IDE.

2.2. Project Dependencies

The next step is to declare the dependencies used by the project in the pom.xml file of the project. If you are not familiar with Maven, do not worry – just replace the contents of the pom.xml file with the listing below.

```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.ivan.springwebflowtut</groupId>
    <artifactId>SpringWebFlowTutorial</artifactId>
    <packaging>war</packaging>
    <version>0.0.1-SNAPSHOT</version>
    <name>SpringWebFlowTutorial JEE5 Webapp</name>

    <properties>
        <!-- Spring framework version used in the project. -->
        <spring.version>3.0.2.RELEASE</spring.version>

        <!-- Spring Web Flow version used in the project. -->
        <springwf.version>2.0.9.RELEASE</springwf.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>com.springsource.javax.servlet.jsp.jstl</artifactId>
            <version>1.1.2</version>
        </dependency>
        <dependency>
            <groupId>org.apache.log4j</groupId>
            <artifactId>com.springsource.org.apache.log4j</artifactId>
            <version>1.2.15</version>
        </dependency>
        <dependency>
```

```
            <groupId>org.jboss.el</groupId>
            <artifactId>com.springsource.org.jboss.el</artifactId>
            <version>2.0.0.GA</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>org.springframework.aop</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>org.springframework.beans</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>org.springframework.context</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>org.springframework.core</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>org.springframework.jdbc</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>org.springframework.orm</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>org.springframework.transaction</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>org.springframework.web</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>org.springframework.web.servlet</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.webflow</groupId>
            <artifactId>org.springframework.faces</artifactId>
            <version>${springwf.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.webflow</groupId>
            <artifactId>org.springframework.js</artifactId>
            <version>${springwf.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.webflow</groupId>
            <artifactId>org.springframework.webflow</artifactId>
            <version>${springwf.version}</version>
        </dependency>

        <!-- Build-time dependencies. -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.1</version>
            <scope>provided</scope>
```

```
        </dependency>

        <!-- Test-time dependencies. -->
        <dependency>
             <groupId>junit</groupId>
             <artifactId>junit</artifactId>
             <version>4.8.1</version>
             <type>jar</type>
             <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.0.2</version>
                <configuration>
                    <!-- Change source and target elements to 1.6 if using Java 1.6. -->
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
        </plugins>
        <finalName>SpringWebFlowTutorial</finalName>
    </build>
</project>
```

Note that:

- Spring 3.0.2 is used.
- Spring Web Flow 2.0.9 is used.
- The dependencies represent a bare minimum required when using Spring Web Flow with JSP display technology.
- The values in the <source> and <target> elements in the Maven compiler plugin configuration should be changed to 1.6 if you are using Java 1.6.

2.3. Web Application Deployment Descriptor

The web application deployment descriptor is located in the WEB-INF directory in the Web Resources node in the Project Explorer. Replace the contents of the web.xml file with the following listing:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
      xsi:schemaLocation="
         http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
      id="WebApp_ID" version="2.5">

      <display-name>SpringWebFlowTutorial</display-name>

      <!-- The welcome file redirects to the entry of the first flow. -->
      <welcome-file-list>
            <welcome-file>index.html</welcome-file>
      </welcome-file-list>

      <!--
        Path to web application's global Spring bean configuration file.
    -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/config/spring-webapp-context.xml</param-value>
    </context-param>

    <!--
        This listener loads the global Spring bean configuration file for the
        web application. Uses the above context parameter
        "contextConfigLocation" to determine the location of the file to load.
    -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
    </listener>

    <!--
        Front Controller of the web application.
    -->
    <servlet>
        <servlet-name>front-controller</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <!--
                Specifies the name and location of the Spring bean
                configuration file local to the front controller.
                The default name is [servlet name]-servlet.xml and
                the default location is in the WEB-INF directory.
            -->
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/config/front-controller-servlet.xml</param-value>
        </init-param>
    </servlet>

    <!--
        Maps all request starting with "flow" to the Front Controller
        servlet.
    -->
    <servlet-mapping>
        <servlet-name>front-controller</servlet-name>
        <url-pattern>/flow/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Note that:

- A welcome file, index.html, is used to direct the application to the first flow when its root URL is accessed.
  We'll see the contents of the welcome file in the next section.
- Two elements, <context-param> and <listener>, and their contents are used to tell Spring to load a bean configuration file from the spring-webapp-context.xml file.
  This bean configuration file contains the Spring beans that are global to the web application and that are inherited by each sub-context (see below).
- The <servlet> and <servlet-mapping> elements and their contents are used to create the front controller servlet of the web application.
  This is the servlet which all requests to the web application passes through. In this web application, it redirects requests associated with flows to resources in a protected directory.
  For details on the Front Controller design pattern, see this webpage.
- The <servlet> element contains an <init-param> element used to specify the bean configuration file containing the beans local to the context of the front controller servlet.
  Further details follow below.

2.4. Welcome Page

As above, the purpose of the welcome page is to redirect requests to the web application root to the entry of the first flow.

- Delete the file index.jsp in the Web Resources node in the Project Explorer.
  This file was automatically generated and we do not need it.
- Create a file at the same location with the name “index.html” with the following contents:

```
<html>
      <head>
        <meta http-equiv="Refresh" content="0; URL=flow/flow1">
      </head>
</html>
```

2.5. Front Controller Bean Configuration File

References: Spring 3.0 Reference, section 15.2.

Each front controller servlet in the Spring framework can have a bean configuration file of its own that contains beans local to the front controller. Please see the reference above for further details!
In this tutorial, the front controller bean configuration file is responsible for loading another bean configuration file which contains the configuration for Spring Web Flow.
Strictly speaking, we could have configured the web.xml file to contain the name of the Spring Web Flow bean configuration file. Using two bean configuration files enables us to add service beans etc. local to the front controller to the bean configuration file below, while keeping the Spring Web Flow configuration in a separate file.

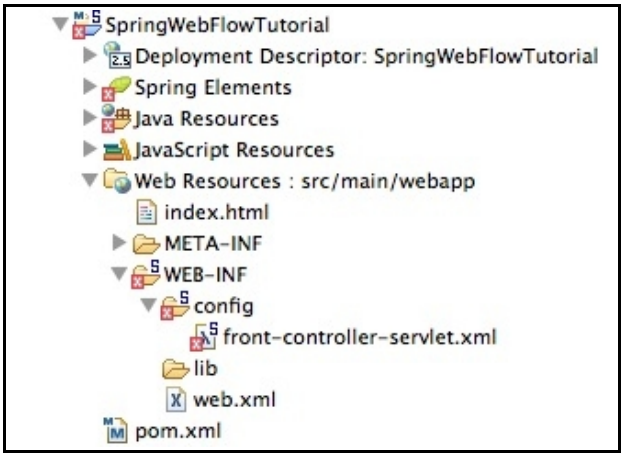
- In the Project Explorer, navigate to the WEB-INF directory in the Web Resources node.
- In the WEB-INF directory, create a directory named “config”.
- In the “config” directory, create a file named “front-controller-servlet.xml” with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

      <!--
         This bean configuration file contains Spring bean context
       configuration local to the front controller context.
       Any beans with the same names inherited from the global
       web application bean context also present in this context
       will be overridden.
      -->

    <!--
       Imports the Spring bean configuration file that contains
       Spring Web Flow beans and configuration.
    -->
    <import resource="webflow-config.xml"/>
</beans>
```

The project structure should, after the above steps have been completed, look like this:



The error is, at this stage, normal and is caused by the missing Spring Web Flow configuration file, something we'll take care of in the next section.

2.6. Spring Web Flow Configuration File

References: Spring Web Flow 2.0.9 Reference, sections 9.4, 9.5.

In the Spring Web Flow bean configuration file of this tutorial, we configure the following things:

- Which flow-definition files to use and their location.
  As we will see, it is also possible to specify a name pattern for the flow-definition files, instead of listing each and every filename.
- The flow executor responsible for executing flows.
  Additional configuration for the flow executor includes listener(s) that supply persistence context(s) for flows the require such and listener(s) that enables the use of Spring Security in combination with Spring Web Flow.

Create a file named “webflow-config.xml” located in the same directory as the front controller bean configuration file created in the previous section, that is in the “WEB-INF/config” directory.

```
<?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"
    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/webflow-config
       http://www.springframework.org/schema/webflow-config/spring-webflow-config-
2.0.xsd">

    <!--
       Register the web flows of the web application.
       We can do this either by:
       1. Listing individual flow files using the <flow-location> element.
       Example: <webflow:flow-location path="/WEB-INF/flows/my-webflow.xml"/>
       2. Register a set of flows matching a pattern we supply using the
       <webflow:flow-location-pattern> element.

       Below, a flow-location-pattern is specified that encourages one
       separate directories for each flow definition.

       The ids of the flows in the web application are affected if a
       base-path is specified for the flow registry.
       The id of a flow will be the remaining path and file name of the
       flow definition file after the base path, minus the file extension.

       It is also possible to have multiple flow registries in one
       web application.
    -->
    <webflow:flow-registry id="flowRegistry" base-path="/WEB-INF/flows">
        <webflow:flow-location-pattern value="**/*-webflow.xml"/>
    </webflow:flow-registry>

    <!--
        The flow executor is responsible for executing flows.
        Listeners that observe the lifecycle of flow executions can be
        registered with the flow executor, for instance:
        1. A listener that manages persistence contexts for flows that
        requires such.
        2. A listener that enables use of Spring Security with Spring
        Web Flow. Listeners can be configured to be applied to
        certain flows.
    -->
    <webflow:flow-executor id="flowExecutor" flow-registry="flowRegistry">
    </webflow:flow-executor>

    <!--
        Adapter enabling Spring Web Flow to run as a controller within
        the front controller. Handles all requests to flows.
        The bean name is used to determine which requests are mapped
        to the controller.
```

```
        The flowExecutor property determines which service is used to
        execute flows.
    -->
    <bean name="/*" class="org.springframework.webflow.mvc.servlet.FlowController">
        <property name="flowExecutor" ref="flowExecutor" />
    </bean>
</beans>
```

Note that:

- A flow registry is configured (bean id “flowRegistry”).
- A base path (“WEB-INF/flows”) is configured for the flow registry.
  Configuring a base path affects the ids of the flows. With a base path configured, the base path is removed from the flow id. The file extension of a flow file is always removed when constructing the id of the flow.
  Example: If a flow is located at “WEB-INF/flows/flow1/mywebflow.xml” and the base path is “WEB-INF/flows”, then the id of the flow becomes “flow1”.
- The flow files to register with the flow registry are specified using a <webflow:flow-location-pattern> element.
  The location pattern we use, “**/*-webflow.xml”, means that the flow file(s) can be located in a directory (relative to the base path) with any name, in a file that ends with “-webflow.xml”.
- An alternative way to specify the flow files to register with the flow registry is to list all of them, using one <webflow:flow-location> element for each file. The path to a flow file is also specified relative to the base path. The id of the flow in the example below becomes “flow1”.
  Example: <webflow:flow-location path="/WEB-INF/flows/flow1/my-webflow.xml"/>
- A flow executor (bean id “flowExecutor”) is configured.
  The flow executor uses the previously configured flow registry.
- Additionally, the following may be configured in the flow executor:
  - Flow execution listeners.
  Example: Listener that manages persistence context(s) for flows, a listener enabling the use of Spring Security with Spring Web Flow.
  Listeners may be configured to only apply to certain flow(s).
  - Flow execution persistence.
  Controlling maximum number of flow executions per user session and controlling maximum number of history snapshots per flow execution.
- The bean with the name “/*” is a controller adapting the Spring Web Flow engine to run as a controller within the front controller.
  The name of the bean is used to determine which requests are mapped to the controller.

2.7. Web Application Global Bean Configuration File

In addition to the bean configuration local to the front controller, we specified a Spring bean configuration file that is to contain the bean definitions global to the web application in the web application deployment descriptor (web.xml).
The file is to be named “spring-webapp-context.xml” and is to be located in the config directory where we placed the web flow configuration file earlier.

```
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
          http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context/spring-context-3.0.xsd">

      <!--
            Scans for annotated classes to be used as beans.
            The base package specified below contains the service beans used
            by the web application.
      -->
      <context:component-scan base-package="com.ivan.springwebflowtut.services" />
</beans>
```

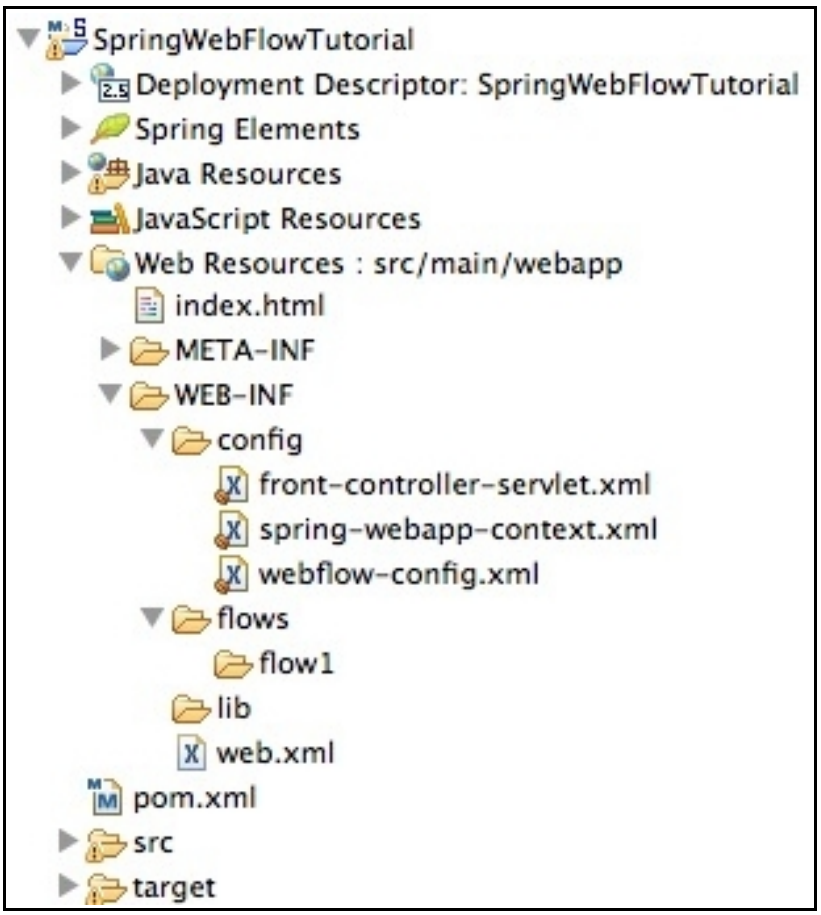
The only thing the above bean configuration file does is, when the file is loaded by Spring, activates a scan that detects annotated classes in a certain package and creates Spring beans accordingly. At the moment, there are no such classes.

2.8. Creating the Flow Directories

The final general step to be performed when setting up a Spring Web Flow project is to create the directories that are to contain the flows. Note that best practices recommends dedicating one directory to store a flow definition and its associated resources.

- In the WEB-INF directory, create a directory named “flows”.
  This is the flow root directory in which one directory for each flow will be created.
- In the “flows” directory, create another directory named “flow1”.
  This is the directory of a flow.

The project structure should now look like this:

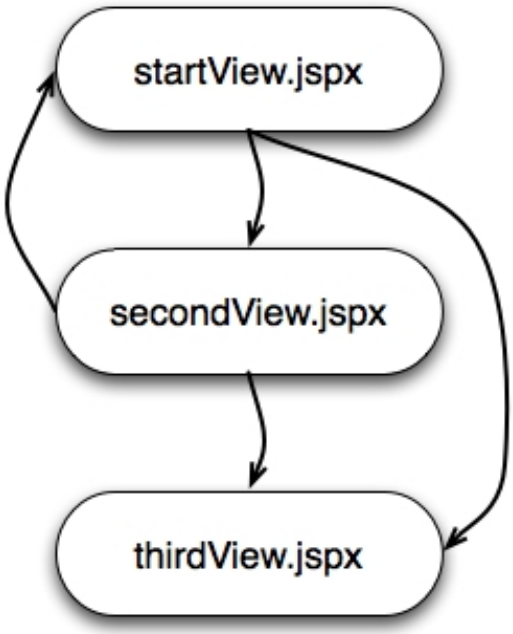


Structure of the project as seen in the Project Explorer.

3. Views, Transitions and Events in Flows

In this chapter we will create our first flow with Spring Web Flow. Remember that a flow is a set of web pages associated with some activity. The web pages are called views. We will also look at Spring Web Flow events and how they affect the transitions between views.

Our first flow consists of three views and has the following structure:



Structure of the first flow, the Hello World flow.

In words, the flow is described as follows:

- There are three views in the flow.
- From the startView, the flow can transition to either the secondView or the thirdView.
- From the secondView, the flow can transition either to the startView or to the thirdView.
- There are no transitions from the thirdView.

With the project set up as described in the previous chapter, all we need to do is to create the three views and a flow definition file.

All views are JSPs written in XML format. I use UTF-8 encoding, so you may have to modify the encoding of these files if you are using a different encoding.

3.1. Creating the Start View

The first view, the start view, is located in a file named “startView.jspx”, which is located in the “flow1” directory created in the previous section. Recall that the flow can transition to either the second or the third view.

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root
    xmlns:jsp="http://java.sun.com/JSP/Page"
    version="2.0">
      <jsp:directive.page language="java"
            contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" />
      <jsp:text>
            <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
      </jsp:text>
      <jsp:text>
            <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
      </jsp:text>

      <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <title>Start View</title>
      </head>
      <body>
        <p>This is the first page of the flow, the Start View.</p>

        <!--
            Note that the method must be specified on the <form> element
            and the value must be "post" in order for Spring Web Flow
            to pick up the event id from the buttons in the form.
        -->
        <form method="post">
            <input type="submit" name="_eventId_toSecond" value="To Second View"/>
            <input type="submit" name="_eventId_toThird" value="To Third View"/>
        </form>
      </body>
      </html>
</jsp:root>
```

Note that:

- The *method* attribute with a value “post” must be specified on the <form> element.
  This specifies the HTTP method with which the form is to be sent when submitted. The default is GET, which will not be recognized by Spring Web Flow.
  Other view technologies, such as Java ServerFaces, has other default values for forms – consult appropriate documentation, if in doubt.
- The <input> elements inside the form have strange names.
  These names specify the Spring Web Flow event ID that will be generated when respective button is clicked. Thus, the first <input> element will generate the “toSecond” Spring Web Flow event and the second <input> element will generate the “toThird” event.
  The underscore characters are significant.
- No *action* attribute is specified in the <form> element.
  The *action* attribute is not needed, since the form will be handled by the Spring Web Flow controller.

3.2. Creating the Second View

The second view is contained in a file named “secondView.jspx” in the “flow1” directory. Recall that the second view can transition either to the first or the third view.

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
      <jsp:directive.page language="java"
            contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" />
      <jsp:text>
            <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
      </jsp:text>
      <jsp:text>
            <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
      </jsp:text>

      <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <title>Second View</title>
      </head>
      <body>
        <p>This is the second page in the flow, the Second View.</p>

        <!--
            Note that the method must be specified on the <form> element
            and the value must be "post" in order for Spring Web Flow
            to pick up the event id from the buttons in the form.
        -->
        <form method="post">
            <input type="submit" name="_eventId_toStart" value="Back To Start" />
            <input type="submit" name="_eventId_toThird" value="To Third View" />
        </form>
      </body>
      </html>
</jsp:root>
```

3.3. Creating the Third View

The third view, which cannot transition to any other view, is located in a file named “thirdView.jspx” in the “flow1” directory.

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
      <jsp:directive.page language="java"
            contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" />
      <jsp:text>
            <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
      </jsp:text>
      <jsp:text>
            <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
      </jsp:text>

      <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <title>Third View</title>
      </head>
      <body>
        <p>This is the third view.</p>
        <p>Hello Web Flow World!</p>
      </body>
      </html>
</jsp:root>
```

3.4. Creating the Flow Definition File

The flow definition file is where all the “magic” takes place. This where:

- A start state of the flow is defined.
- The views of the flow are configured.
- We define which events Spring Web Flow is to react to during the presentation of certain views.
- Possible transition(s) between views.
- One or more end states of the flow are defined.

The flow definition file is named “flow1-webflow.xml” and located in the “flow1” directory. Recall that we configured the name pattern “**/*-webflow.xml” for flow definition files when we created the Spring Web Flow configuration file above.

```
<?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"
   start-state="start">
   <!--
        Note that the start-state is specified using an attribute of
        the <flow> element above.
    -->

    <!--
        A view state is a step of the flow that renders a view.
        This state displays the startView and configures the two available
        transitions from the startView:
        - On receiving the toSecond event, transition to the "second" state.
        - On receiving the toThird event, transition to the "third" state.
    -->
      <view-state id="start" view="startView.jspx">
         <transition on="toSecond" to="second"/>
         <transition on="toThird" to="third"/>
      </view-state>

    <!--
        Similar to the above view state, but for the secondView.
    -->
    <view-state id="second" view="secondView.jspx">
        <transition on="toStart" to="start"/>
        <transition on="toThird" to="third"/>
    </view-state>

    <!--
        The final state of the flow in which an outcome of the flow
        can be defined.
        A flow may have multiple <end-state> elements.
    -->
    <end-state id="third" view="thirdView.jspx"/>
</flow>
```

Note that:

- The root element of the flow definition file is the <flow> element.
- The start state of the flow is defined using the *start-state* attribute in the <flow> element.
- A view state is described using the <view-state> element.
- A view state has an id and an associated view that is rendered when entering the view state.
- A view state can have zero or more transitions, specifying which event(s) trigger transition to which view state.
- A transition is specified using the <transition> element.
  The value of the *on* attribute specify the event that triggers the transition and the *to* attribute specifies the id of the view state which is the target of the transition.
- The final state of the flow is specified using the <end-state> element.
- A flow may contain multiple <end-state> elements.

3.5. Running the Application

At this stage we are now read to run our web application.

- Right-click the project in the Project Explorer and select Run As -> Run on Server.
- Select the Tomcat server on which to run the web application.
- When the server has started and the application is deployed, try clicking the different buttons and see if the flow indeed works as expected.

4. Spring Web Flow Logging

In this chapter we will take a look at the log output generated by Spring Web Flow. Spring Web Flow has in fact several different lo categories that can be enabled and disabled as desired. Temporarily enabling one or more log categories during debugging can be very helpful. Being selective in which categories to enable is a good idea, since the amount of log generated quickly becomes overwhelming.

Spring Web Flow uses the Log4J framework for logging and has, as far as I have been able to discover, the following categories:

Log4J Category	Description
log4j.category.org.springframework.webflow	Core Spring Web Flow log.
log4j.category.org.springframework.binding	Log related to Spring Web Flow's data binding.
log4j.category.org.springframework.faces	Log related to integration with Java ServerFaces.
log4j.category.org.springframework.transaction	Log related to transactions in connection to flow managed persistence.
log4j.category.org.springframework.js	Log related to resource retrieval from JAR files.

We will use the example project developed in chapters 2 and 3 and add logging to examine, among other things, the events generated in the different views, the state of the conversation etc.

4.1. Creating the Log Configuration File

The log configuration file is a standard Log4J configuration file named “log4j.properties”.

- If not already present, create a new source folder “src/main/resources” under the Java Resources node in the Project Explorer.
- In the “src/main/resources” source folder, create a new file named “log4j.properties” with the following contents:

```
# Log4J configuration file.
log4j.rootCategory=OFF

# Enable core web flow logging.
log4j.category.org.springframework.webflow=DEBUG, stdout

# An log appender that writes log to the console.
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - <%m>%n
```

4.2. Examining Spring Web Flow Log

If we run the web application with the log configuration file in place, there will be a significant increase in log output to the console. We will take a look at some events that are logged.

When the web application starts up, it registers the flow definitions it finds (since we used a flow location pattern in the flow configuration file). The following line shows that a flow definition file has been discovered and registered with the flow id "flow1" (the name of the flow directory):

```
2010-05-18 17:38:39,388 DEBUG
[org.springframework.webflow.definition.registry.FlowDefinitionRegistryImpl] -
<Registering flow definition 'ServletContext resource [/WEB-INF/flows/flow1/flow1-
webflow.xml]' under id 'flow1'>
```

Further down in the log, we can see what state the flow enters:

```
2010-05-18 17:38:40,158 DEBUG [org.springframework.webflow.engine.ViewState] - <Entering
state 'start' of flow 'flow1'>
```

If we click the button "To Second View" on the first web page, we can see the Spring Web Flow event generated (in this case "toSecond"):

```
2010-05-18 17:46:00,376 DEBUG [org.springframework.webflow.engine.ViewState] - <Event
'toSecond' returned from view [ServletMvcView@1a2e34bf view =
org.springframework.web.servlet.view.JstlView: unnamed; URL [/WEB-
INF/flows/flow1/startView.jspx]]>
```

A couple of lines further down we can see the transition caused by the event:

```
2010-05-18 17:49:23,588 DEBUG [org.springframework.webflow.engine.Transition] -
<Executing [Transition@4f88f506 on = toSecond, to = second]>
```

Finally, when clicking the button "To Third View" on either the first or second web page, we can see that the execution of the flow ends (this since the third flow state is an end state):

```
2010-05-18 17:51:20,335 DEBUG [org.springframework.webflow.engine.Transition] -
<Completed transition execution.  As a result, the flow execution has ended>
2010-05-18 17:51:20,335 DEBUG
[org.springframework.webflow.execution.repository.impl.DefaultFlowExecutionRepository] -
<Removing flow execution '[Ended execution of 'flow1']' from repository>
```

This concludes the brief look at the log generated by Spring Web Flow.

5. Unit Testing Flows - Basics

Spring Web Flow also provides means to test web flows using unit testing style tests, including the ability to mock subflows and services the flow depends on. In this chapter we will write a test for the flow developed in chapter 3.
Spring Web Flow provides a class, *AbstractXmlFlowExecutionTests*, that flow unit test classes can subclass, in order to make developing flow unit tests easier. This class, in turn, is a subclass of the JUnit class *TestCase*, thus flow unit tests are JUnit tests.

There are two options regarding the example project of this chapter:

- If you have the example program from chapter three at hand, then you can continue to use that project.
- If not, then you have to start by setting up a project as described in chapter two and then create the flow definition file, as previously described.

In addition to this chapter, there is an additional example, including addition aspects, such as mocking of services, in chapter 10.

5.1. Creating the Flow Unit Test

In the project node “src/test/java”, in the package *com.ivan.springwebflowtut*, create a class *Flow1Test* that is a subclass of *AbstractXmlFlowExecutionTests*. The IDE will automatically add an implementation of the abstract method *getResource*. This method is to retrieve the flow resource of the flow to be tested.

5.2. Retrieving the Flow To Be Tested

In the *getResource* method, we'll use the enclosed resource factory to retrieve the flow resource of the flow to be tested. The complete test class so far looks like this:

```
package com.ivan.springwebflowtut;

import org.springframework.webflow.config.FlowDefinitionResource;
import org.springframework.webflow.config.FlowDefinitionResourceFactory;
import org.springframework.webflow.core.collection.LocalAttributeMap;
import org.springframework.webflow.core.collection.MutableAttributeMap;
import org.springframework.webflow.test.MockExternalContext;
import org.springframework.webflow.test.execution.AbstractXmlFlowExecutionTests;

/**
 * This class implements tests of the "flow1" Spring Web Flow flow.
 * A flow test verifies that a flow executes as it is supposed to,
 * given certain input etc.
 * A flow test case inherits from the JUnit <code>TestCase</code>
 * class, so this class is a JUnit test case.
 *
 * @author Ivan A Krizsan
 */
public class Flow1Test extends AbstractXmlFlowExecutionTests
{
    /* Constant(s): */
    private final static String FLOW1_FILE_LOCATION =
        "src/main/webapp/WEB-INF/flows/flow1/flow1-webflow.xml";
    private final static String FLOW1_INITIAL_STATE = "start";

    /**
     * Retrieves the flow resource for the flow to test using the
     * supplied resource factory.
     *
```

```
     * @param inResourceFactory Factory used to create flow resource.
     * @return Flow resource of flow to test.
     */
    @Override
    protected FlowDefinitionResource getResource(
        final FlowDefinitionResourceFactory inResourceFactory)
    {
        FlowDefinitionResource theFlowResource;

        theFlowResource =
            inResourceFactory.createFileResource(FLOW1_FILE_LOCATION);
        return theFlowResource;
    }
}
```

5.3. Testing Starting the Flow

The next step is to test the flow startup and make sure that it enters its initial state. We will first make a deliberate error in the test to become acquainted with failing tests.

- Add the following class constant to the *Flow1Test* class:

```
private final static String FLOW1_INITIAL_STATE = "end";
```

- Add the following test method to the *Flow1Test* class:

```
    /**
     * Tests starting the flow and the initial state of the flow.
     */
    public void testFlowStartup()
    {
        /* Attribute map that contains input parameters to the flow. */
        MutableAttributeMap theFlowInput = new LocalAttributeMap();

        /*
         *  Mock version of the object used to access the Spring Web Flow
         *  client environment.
         */
        MockExternalContext theMockExternalContext = new MockExternalContext();

        /* Start the execution of the flow to be tested. */
        startFlow(theFlowInput, theMockExternalContext);

        /*
         * Make sure that the current state of the flow, after the
         * startup, is the initial state of the flow.
         */
        assertCurrentStateEquals(FLOW1_INITIAL_STATE);
    }
```

Note that:

- The test class keeps track of the flow-under-test for us.
- Input to the flow is supplied in a map.
  In this case, we supply an empty map, since there is no input to the flow.
- When starting the flow, we also supply a mock external context.
  Again, this is not used by our flow. More about mocking later in this tutorial.
- To start the flow, we call the *startFlow* method of the test class, supplying the input map and the mock external context as parameters.
- The current state of the flow can be asserted using the *assertCurrentStateEquals* method of the test class.

There are a number of additional assert methods related to unit testing of flows supplied by the parent class of the test class. Use content assist in the editor to find them (commonly ctrl-space)!

Switch to the Java Perspective in the IDE, if you are not there already, and run the test. The test should fail and a red bar should appear in the JUnit view. The first part of the failure trace looks like this:

```
junit.framework.ComparisonFailure: The current state 'start' does not equal the expected
state 'end' expected:<[end]> but was:<[start]>
      at junit.framework.Assert.assertEquals(Assert.java:81)
      at
org.springframework.webflow.test.execution.AbstractFlowExecutionTests.assertCurrentState
Equals(AbstractFlowExecutionTests.java:332)
      at com.ivan.springwebflowtut.Flow1Test.testFlowStartup(Flow1Test.java:68)
      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
      at
...
```

From this we can see that the state of the flow did not match the state supplied in the assertion.
If we modify the class constant to have the value “start”, like below, the test should now pass when being re-run.

```
private final static String FLOW1_INITIAL_STATE = "start";
```

5.4. Testing Flow Transitions - Basics

We also want to test that the flow responds to events by making the appropriate transitions from one state to another.

- Add the following class constants to the *Flow1Test* class:

```
    private final static String FLOW1_SECOND_STATE = "second";
    private final static String FLOW1_THIRD_STATE = "third";
    private final static String FLOW1_TOFIRST_EVENT = "toStart";
    private final static String FLOW1_TOSECOND_EVENT = "toSecond";
    private final static String FLOW1_TOTHIRD_EVENT = "toThird";
```

- Add the following test method to the *Flow1Test* class:

```
    /**
     * Tests the transition of the flow from the first state to the
     * second state.
     */
    public void testFirstToSecondTransition()
    {
        /*
         * In this test we will actually use the mock external context,
         * having it simulate a user-generated event.
         */
        MockExternalContext theMockExternalContext = new MockExternalContext();

        /*
         * Set the event id that the flow is to receive in the
         * external context mock object.
         */
        theMockExternalContext.setEventId(FLOW1_TOSECOND_EVENT);

        /* Set the current flow state to the start state. */
        setCurrentState(FLOW1_INITIAL_STATE);

        /*
         * Resume execution of the flow.
         * The flow will react to the event set above and is then
         * expected to transition to a new state.
         */
```

```
        resumeFlow(theMockExternalContext);

        /*
         * Make sure that the flow has transitioned to the expected
         * state as a result of the event.
         */
        assertCurrentStateEquals(FLOW1_SECOND_STATE);
    }
```

Note that:

- A mock external context is created, like in the previous test method, but in this test method we call the *setEventId* method on the mock context to simulate an user generating an event.
- To resume the execution of the flow and allow it to react to the event, we call the *resumeFlow* method in the test class.

If we run the test, we see that it passes and, thus, when in the first state and receiving the "toSecond" event, the flow will indeed transition to the second state.

5.5. Testing Flow Transitions – To End of Flow

Now we set out to test how the flow, being in the first state and receiving the event indicating that it should transition to the third state, reacts. A first attempt at implementing the test is done by adding the following method to the *Flow1Test* class:

```
    /**
     * Tests the transition of the flow from the first state to the
     * third state.
     */
    public void testFirstToThirdTransition()
    {
        MockExternalContext theMockExternalContext = new MockExternalContext();
        theMockExternalContext.setEventId(FLOW1_TOTHIRD_EVENT);
        setCurrentState(FLOW1_INITIAL_STATE);
        resumeFlow(theMockExternalContext);
        assertCurrentStateEquals(FLOW1_THIRD_STATE);
    }
```

If we execute the test case, the new test method fails. The first part of the failure trace looks like this:

```
java.lang.IllegalStateException: No active FlowSession to access; this FlowExecution has
ended
      at
org.springframework.webflow.engine.impl.FlowExecutionImpl.getActiveSession(FlowExecution
Impl.java:191)
      at
org.springframework.webflow.test.execution.AbstractFlowExecutionTests.assertCurrentState
Equals(AbstractFlowExecutionTests.java:332)
      at
com.ivan.springwebflowtut.Flow1Test.testFirstToThirdTransition(Flow1Test.java:122)
      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
      at
...
```

Alas, our flow is no more and thus we cannot assert that it has entered the third state!

To our rescue, there is another assert method, *assertFlowExecutionEnded*. Replacing the last assertion with a call to the new assert method, the test method now looks like this:

```
    public void testFirstToThirdTransition()
    {
        MockExternalContext theMockExternalContext = new MockExternalContext();
        theMockExternalContext.setEventId(FLOW1_TOTHIRD_EVENT);
        setCurrentState(FLOW1_INITIAL_STATE);
        resumeFlow(theMockExternalContext);
        assertFlowExecutionEnded();
    }
```

If we re-run the test, it now passes.

5.6. Testing Flow Transitions – Illegal Transitions

In the next test we want to test how the flow, being in the first state and receiving the event indicating that it should transition to the first state, reacts. A first attempt at implementing the test is done by adding the following method to the *Flow1Test* class:

```
    /**
     * Tests the illegal case when the flow being in the first state
     * receives an event indicating that it should transition
     * to the first state.
     */
    public void testFirstToFirstTransition()
    {
        MockExternalContext theMockExternalContext = new MockExternalContext();
        theMockExternalContext.setEventId(FLOW1_TOFIRST_EVENT);
        setCurrentState(FLOW1_INITIAL_STATE);
        resumeFlow(theMockExternalContext);
        assertCurrentStateEquals(FLOW1_INITIAL_STATE);
    }
```

If we run this test, it fails because there is no transition from the first state of the flow to the first state of the flow. How do we verify an illegal transition?

First, lets take a look at the first part of the failure trace for the original test method:

```
org.springframework.webflow.engine.NoMatchingTransitionException: No transition found on
occurence of event 'toStart' in state 'start' of flow 'flow1-webflow' -- valid
transitional criteria are array<TransitionCriteria>[toSecond, toThird] -- likely
programmer error, check the set of TransitionCriteria for this state
      at
org.springframework.webflow.engine.TransitionableState.getRequiredTransition(Transitiona
bleState.java:93)
      at
org.springframework.webflow.engine.TransitionableState.handleEvent(TransitionableState.j
ava:119)
      at org.springframework.webflow.engine.Flow.handleEvent(Flow.java:555)
      at
org.springframework.webflow.engine.impl.FlowExecutionImpl.handleEvent(FlowExecutionImpl.
java:386)
      at
...
```

We can see that a *NoMatchingTransitionException* is thrown and modify our test method to expect such an exception.

The new version of the test method looks like this:

```
    public void testFirstToFirstTransition()
    {
        boolean theExceptionFlag = false;
        MockExternalContext theMockExternalContext = new MockExternalContext();
        theMockExternalContext.setEventId(FLOW1_TOFIRST_EVENT);
        setCurrentState(FLOW1_INITIAL_STATE);

        /*
         * There should be an exception thrown when resuming the flow,
         * since there is no transition from the first state to the first
         * state.
         */
        try
        {
            resumeFlow(theMockExternalContext);
        } catch (final NoMatchingTransitionException theException)
        {
            theExceptionFlag = true;
        }
        assertTrue(theExceptionFlag);
    }
```

Running the test case, all the tests now pass.

This concludes the first look at unit testing of web flows.

6. Invoking Service Beans from Flows

Web pages usually present some kind of data to the user of the web page. Quite often, this data is dynamically generate; for instance, as a result of a search I want to see the matching entries, not all entries.
In this chapter we will look at how to invoke service beans from a Spring Web Flow, in order to retrieve data to present to the user of our web application. The example web application consists of one single web page only, which lists some data retrieved from a service bean.

The base for the example program in this section is a project set up as described in chapter two.

6.1. Creating Service Bean

First, we'll implement the service bean that is to supply the data to be listed in the view. The service will generate a list of a random number, from one to ten, of strings in an array.
Before implementing the bean, create a package in the “src/main/java” node in the Java Resources node. The package name is to be “com.ivan.springwebflowtut.services”. The bean is implemented as follows:

```
package com.ivan.springwebflowtut.services;

import org.springframework.stereotype.Service;

/**
 * Implements the service backing the start view in the flow flow1.
 * This service is considered to be in the view layer and may thus contain
 * view-technology specific references.
 * Its main purpose should be to act as an entrypoint to the services in
 * the service layer, which should not be directly accessed from the
 * view layer.
 * This class may implement the following design patterns:
 * - Facade
 * - Front Controller
 * - Adapter
 *
 * @author Ivan A Krizsan
 */
@Service("startViewService")
public class StartViewService
{
    /**
     * Retrieves a list of entries that is to be presented to the user.
     *
     * @return List of entries.
     */
    public String[] retrieveList()
    {
        /* Slightly random number of items each time. */
        int theCount = (int)(System.currentTimeMillis() % 10) + 1;
        String[] theResultList = new String[theCount];
        for (int i = 0; i < theCount; i++)
        {
            theResultList[i] = "Result Entry " + (i + 1);
        }

        return theResultList;
    }
}
```

Note that:

- The *StartViewService* class is annotated with the @Service annotation.
  With our Spring bean configuration file appropriately configured, this will cause a bean to be created having the name supplied in the annotation.

6.2. Creating the Flow Definition File

Apart from the flow configuration we have seen in previous examples, the service bean implemented in the previous section will be invoked from the flow definition file. Data is placed in a variable that is accessible in the view.
The flow definition file is named “flow1-webflow.xml” and located in the “flow1” directory.

```
<?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"
    start-state="start">

    <view-state id="start" view="startView.jspx">
        <!--
            The <on-render> element allows evaluation of one or more
            expressions prior to the rendering of the view.
            In this example, we retrieve the data to be presented in
            the view here.
            The result of the evaluation of the expression is put
            in the variable 'resultList', which lasts for the entire
            conversation (in the conversationScope).
        -->
        <on-render>
            <!--
                Using the <evaluate> element, the method retrieveList
                is invoked on the startViewService Spring bean.
                The result is placed in the variable resultList in the
                conversation scope.
            -->
            <evaluate expression="startViewService.retrieveList()"
                result="conversationScope.resultList"/>
        </on-render>

        <!--
            This transition enables refreshing the view and thus also
            the list of entries in the view.
        -->
        <transition on="toFirst" to="start">
        </transition>
    </view-state>
</flow>
```

Note that:

- In the <view-state> element, there is an <on-render> element.
  The <on-render> element enables evaluation of one or more expressions prior to the rendering of the view.
- In the <on-render> element, there is a <evaluate> element.
  The attribute *expression* holds the expression that is to be evaluated. In this example, the method *retrieveList* on the Spring bean *startViewService* is to be invoked.
  The optional attribute *result* tells Spring Web Flow that the result is to be placed in the variable *resultList* in the conversation scope.
  More about Spring Web Flow variable scopes in a subsequent chapter.

6.3. Creating the View

The start view, which is also the only view in this chapter's example, is located in a file named "startView.jspx", which is located in the "flow1" directory created in the previous section.

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root
    xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:fn="http://java.sun.com/jsp/jstl/functions"
    version="2.0">
      <jsp:directive.page language="java"
            contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" />
      <jsp:text>
            <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
      </jsp:text>
      <jsp:text>
            <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
      </jsp:text>

      <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
            <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
            <title>Start View</title>
      </head>
      <body>
            <p>This is the first page of the flow, the Start View.</p>
            These are the current entries:<br/>

            <form method="post">
                <ul>
                    <c:forEach items="${resultList}" var="item">
                        <li>
                            ${item}
                        </li>
                    </c:forEach>
                </ul>

                <input type="submit" name="_eventId_toFirst" value="Retrieve Entries"/>
            </form>
      </body>
      </html>
</jsp:root>
```

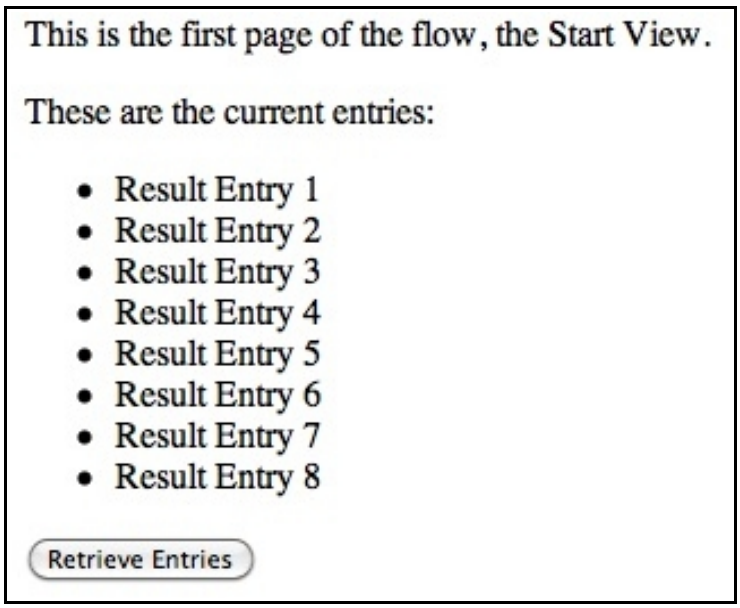
Note that:

- The expression ${resultList} is used in the <c:forEach> tag to access the list of items to be presented to the user.
  The JSP technology knows nothing about the Spring Web Flow scopes, so we just supply the name of the variable to access its contents.

6.4. Running the Application

We are now ready to run the example web application created in this chapter.

- Right-click the project in the Project Explorer and select Run As -> Run on Server.
- Select the Tomcat server on which to run the web application.
- The start view should appear, with a list similar to the one in the figure and one button:



The first and only web page of this chapter's example web application.

- For each click on the “Retrieve Entries” button, the list will be updated.
  Note that if the list contains the exact same number of items, then it will appear as if nothing has happened.

This concludes this chapter's example program. We will look more closely at the different times in the life-cycle of a flow when service beans can be invoked in chapter eight.

7. Error Handling in Flows

Exceptions may occur during the execution of an application even though it uses Spring Web Flow. Most often, we do not want to present stack traces and similar things to the user, but rather show an error page.
With Spring Web Flow, exceptions can act as events and cause transitions to take place depending on the kind of exception that occurs. In this chapter we will take a look at how to handle exceptions occurring either in Spring Web Flow or in underlying service beans. In this chapter we'll see how to handle exceptions using only the flow definition file. In subsequent chapters more advanced exception handling will be examined.

The base for the example program in this section is a project set up as described in chapter two.

7.1. Creating the Start View

The first view, the start view, is located in a file named “startView.jspx”, which is located in the “flow1” directory.

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root
    xmlns:jsp="http://java.sun.com/JSP/Page"
    version="2.0">
      <jsp:directive.page language="java"
            contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" />
      <jsp:text>
            <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
      </jsp:text>
      <jsp:text>
            <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
      </jsp:text>

      <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <title>Start View</title>
      </head>
      <body>
        <p>This is the first page of the flow, the Start View.</p>

        <form method="post">
            <input type="submit" name="_eventId_toSecond" value="To Second View"/>
            <input type="submit" name="_eventId_toThird" value="To Third View"/>
            <input type="submit" name="_eventId_toIllegal" value="Illegal Transition"/>
        </form>
      </body>
      </html>
</jsp:root>
```

The three buttons on the above webpage will cause three different types of exceptions to be throw:

- The first will cause a checked exception to be thrown from a service bean.
- The second will cause an unchecked exception to be thrown from a service bean.
- The third button will cause an exception caused by an illegal transition to be thrown from Spring Web Flow.

7.2. Creating the Error Template Page

A template JSP will be used for the error reporting web page. The template is located in the root of the Web Resources node (src/main/webapp) in the Project Explorer in a file named "errorTemplate.jsp".

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <title>An Error Occurred</title>
    </head>
    <body>
        <h3>An error occurred:</h3>
        ${param.errorMessage}
    </body>
</html>
```

The error template page is placed outside of all flow directories, in order to be usable by all flows.

7.3. Creating the Checked Exception Error View

The checked exception error view is used to report a checked exception having occurred. This view will be used in the flow, so it must be located in the flow directory previously created ("flow1"). The name of the file is "checkedError.jspx".

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root
    xmlns:jsp="http://java.sun.com/JSP/Page"
    version="2.0">
    <jsp:directive.page
        language="java"
        contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8" />
    <jsp:text>
        <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
    </jsp:text>
    <jsp:text>
        <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
    </jsp:text>
    <jsp:include page="/errorTemplate.jsp">
        <jsp:param
            name="errorMessage"
            value="Oops, an error caused by a checked exception occurred!" />
    </jsp:include>
</jsp:root>
```

7.4. Creating the Unchecked Exception Error View

The unchecked exception error view is used to report unchecked exceptions. Again, this view will be used in the flow, so it must be located in the flow directory previously created ("flow1").
The name of the file is "uncheckedError.jspx".

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root
    xmlns:jsp="http://java.sun.com/JSP/Page"
    version="2.0">
    <jsp:directive.page
        language="java"
        contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8" />
    <jsp:text>
        <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
    </jsp:text>
    <jsp:text>
        <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
    </jsp:text>
    <jsp:include page="/errorTemplate.jsp">
        <jsp:param
            name="errorMessage"
            value="Oops, an error caused by an unchecked exception occurred!" />
    </jsp:include>
</jsp:root>
```

With all the view-related files created, we now go on to create the single service bean used in this example.

7.5. Creating the Exception-Generating Service Bean

One of the things we are interested in is exceptions thrown from service beans. That is, Spring beans that are invoked from views in the flow. Before being able to implement the bean, create a package in the "src/main/java" node in the Java Resources node. The package name is to be "com.ivan.springwebflowtut.services". The bean is implemented as follows:

```
package com.ivan.springwebflowtut.services;

import java.util.Date;
import org.springframework.stereotype.Service;

/**
 * Implements the service backing the start view in the flow1 flow.
 * This service generates checked and unchecked exceptions.
 *
 * @author Ivan A Krizsan
 */
@Service("startViewService")
public class StartViewService
{
    public void generateCheckedException() throws Exception
    {
        Date theCurrentTime = new Date();
        throw new Exception("Checked exception at " + theCurrentTime);
    }

    public void generateUncheckedException() throws Exception
    {
        Date theCurrentTime = new Date();
        throw new Error("Unchecked exception at " + theCurrentTime);
    }
}
```

7.6. Creating the Flow Definition File

In this example program, the flow definition file is where we declare how exceptions are to be handled. The flow definition file is named “flow1-webflow.xml” and located in the “flow1” directory.
Compared to previous example, the flow definition file is incomplete in that it does not contain all the views and it does not contain an end state – this is intentional, in order to minimize configuration not immediately related to error handling.

```
<?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"
    start-state="start">

    <view-state id="start" view="startView.jspx">
        <transition on="toSecond" to="second">
            <evaluate expression="startViewService.generateCheckedException()"/>
        </transition>
        <transition on="toThird" to="third">
            <evaluate expression="startViewService.generateUncheckedException()"/>
        </transition>
    </view-state>

    <!--
        Error handling view state to be shown when an error caused
        by a checked exception occurs.
    -->
    <view-state id="checkedError" view="checkedError.jspx">
    </view-state>

    <!--
        Error handling view state to be shown when an error caused
        by an unchecked exception occurs.
    -->
    <view-state id="uncheckedError" view="uncheckedError.jspx">
    </view-state>

    <!--
        Transitions global to all the states in the flow.
        Will be executed if the current state does not contain a
        matching transition.
    -->
    <global-transitions>
        <!--
            Transition to be made if an unchecked exception occurs.
            Will also invalidate history to prevent backtracking
            to any previously displayed view.
        -->
        <transition on-exception="java.lang.Error" to="uncheckedError"
            history="invalidate">
        </transition>
        <!--
            Transition to be made if a checked exception occurs.
            Will also invalidate history to prevent backtracking
            to any previously displayed view.
        -->
        <transition on-exception="java.lang.Exception" to="checkedError"
            history="invalidate">
        </transition>
    </global-transitions>
</flow>
```

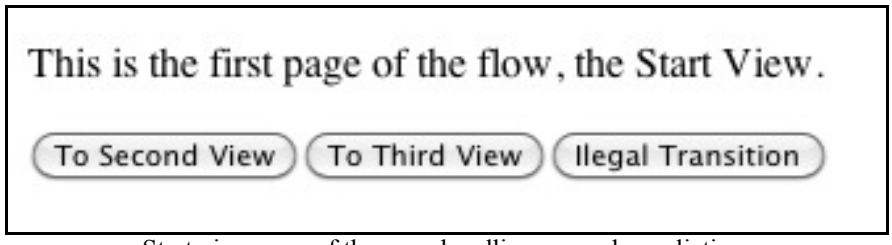
Note that:

- There are two view states, checkedError and uncheckedError, which are the error states to which the flow will transition when a checked respective an unchecked exception occurrs.
- There is a <global-transition> element.
  This element contains transitions that can be executed anywhere in the flow, provided that the current state does not contain a matching transition.
  In this case we want the error handling to apply to all the states in the flow, so we place the error-handling transitions in a <global-transition> element.
- In the <transition> elements, in the <global-transitions> element, we can see two attributes we haven't seen before: The *on-exception* attribute and the *history* attribute.
  The *on-exception* attribute specifies the kind of exception that triggers the transition.
  The *history* attribute, which default value is "preserve", determines how the history used for view backtracking is to be maintained.
  The value "preserve" means that that the user, from the next view, is to be able to backtrack to the previous view and all other views in the history.
  The value "discard" means that the user, from the next view, is going to be unable to backtrack to the previous view but be able to backtrack to other views in the history.
  Finally, the value "invalidate" means that the user will not be able to backtrack to any view.

7.7. Running the Application

Now we are ready to run our web application.

- Right-click the project in the Project Explorer and select Run As -> Run on Server.
- Select the Tomcat server on which to run the web application.
- The start view should appear, with three buttons on the web page:



Start view page of the error handling example appliation.

- Click the button "To Second View".
  Another web page with the message "Oops, an error caused by a checked exception occurred!" should appear.
  If we examine the console log, we should be able to see an exception stack trace with the root cause being:
  ```
  Caused by: java.lang.Exception: Checked exception at Thu May 20 18:30:29 CEST 2010 at
  com.ivan.springwebflowtut.services.StartViewService.generateCheckedException
  (StartViewService.java:32)
  ```
  The time will, of course, differ, but we can also see that the exception originates from our *StartViewService* class.
- Restart the application and click the "To Third View" button on the first page.
  Again, an web page indicating an error should occur, but this time the message is "Oops, an error caused by an unchecked exception occurred!".
  In the console log, we can see the root cause:
  ```
  Caused by: java.lang.Error: Unchecked exception at Thu May 20 18:36:18 CEST 2010 at
  ```

```
com.ivan.springwebflowtut.services.StartViewService.generateUncheckedException
(StartViewService.java:39)
```
Again, the time will differ, but we can see that the exception originates from our *StartViewService* class.

- For the last time, restart the application but now click the “Illegal Transition” button on the first page.
  Again, an web page indicating an error should occur, but this time the message is “Oops, an error caused by a checked exception occurred!”.
  In the console log, we can see the root cause:
  ```
  2010-05-20 18:41:18,202 DEBUG
  [org.springframework.webflow.engine.support.TransitionExecutingFlowExecutionExceptionHandler
  ] - <Handling flow execution exception
  org.springframework.webflow.engine.NoMatchingTransitionException: No transition found on
  occurence of event 'toIllegal' in state 'start' of flow 'flow1' -- valid transitional
  criteria are array<TransitionCriteria>[toSecond, toThird] -- likely programmer error, check
  the set of TransitionCriteria for this state>
  org.springframework.webflow.engine.NoMatchingTransitionException: No transition found on
  occurence of event 'toIllegal' in state 'start' of flow 'flow1' -- valid transitional
  criteria are array<TransitionCriteria>[toSecond, toThird] -- likely programmer error, check
  the set of TransitionCriteria for this state at
  org.springframework.webflow.engine.TransitionableState.getRequiredTransition(TransitionableS
  tate.java:93)
  ```
  This time, we see that the exception was thrown from within the Spring Web Flow framework and that the cause is an attempt to transition to a non-existing state.
  The cause for this is that the third button generates an event for which there is no transition define in the flow.

This concludes the first example on error handling.

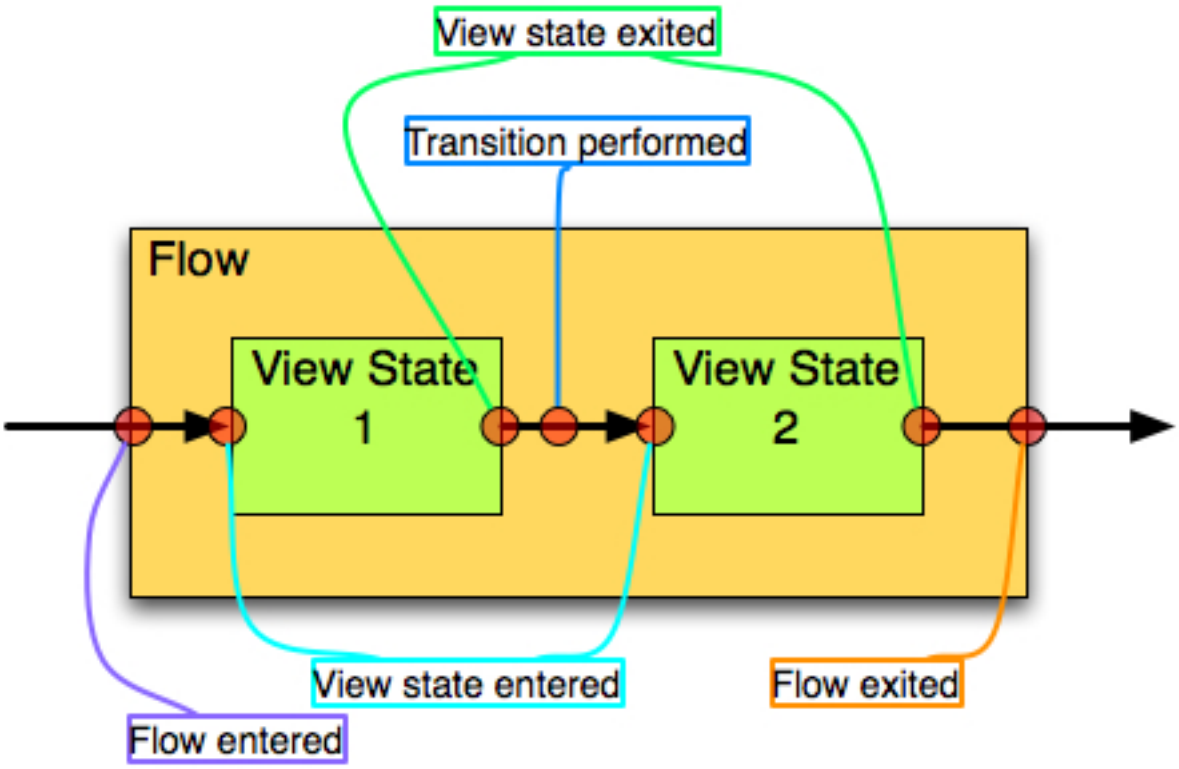
TODO remove the history clearing from the above example and put in separate chapter?

TODO check all links in document

8. Invoking Service Beans and the Flow Lifecycle

In chapter six, we saw how service beans can be invoked from a Spring Web Flow flow. The example showed how an array of data was retrieved immediately prior to the view being rendered. Service beans, as well as other actions, can be undertaken at different times in the lifecycle of a flow:

- As a flow is entered.
- As a flow is exited.
- As a view/flow state is entered.
- As a view/flow state is exited.
- As a transition is performed.



Points in the lifecycle of a flow that actions can be invoked at.

In this chapter's example program, we will invoke different methods in a service bean at all the different points in the life-cycle of a flow. As in the above picture, our web application will only have two view-states and one single transition between these states.
In addition to the two views in the flow, we will also have one web page prior to the flow, from which the flow is entered, and another web page immediately after the flow, which is shown when the flow has ended.

As usual, we first create a project for the example web application as described in chapter two.

8.1. Creating the Webpages Show Before and After the Flow

First, we'll create the web pages that are to be shown before entering the Spring Web Flow flow in the example web application. Both web pages are to be located in the Web Resources root in the project.

Creating the Before Flow Page

The page to be shown before entering the flow is stored in a file named “before-flow.xhtml”:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <title>Before the Flow</title>
    </head>
    <body>
        <h3>This page is show before the flow is entered.</h3>
        <p>
            <a href="flow/flow1">Enter the Flow</a>
        </p>
    </body>
</html>
```

Note that:

- The link used to enter the flow is a relative link to the root of the “flow1” flow.

Creating the After Flow Page

Consequently, the page to be shown after having exited the flow is stored in a file named “after-flow.xhtml”:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <title>After the Flow</title>
    </head>
    <body>
        <h3>The flow has now finished, this web page does not belong to the flow.</h3>
    </body>
</html>
```

8.2. Modifying the Welcome Page

The welcome page created when setting up the project will redirect to the first page in the flow. In this example, this is not what we want, so we need to modify the welcome page in the file “index.html” to redirect to the before-flow page instead. The modified “index.html” looks like this:

```
<html>
    <head>
      <meta http-equiv="Refresh" content="0; URL=before-flow.xhtml">
    </head>
</html>
```

8.3. Creating the First Flow View

The first flow view is, again, in the file “startView.jspx” in the “flow1” directory. It contains one single button allowing the user to transition to the second view.

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
    <jsp:directive.page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"/>
    <jsp:text>
        <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
    </jsp:text>
    <jsp:text>
        <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
    </jsp:text>

    <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <title>Start View</title>
    </head>
    <body>
        <p>This is the first page of the flow, the Start View.</p>

        <form method="post">
            <input type="submit" name="_eventId_toSecond" value="To Second View"/>
        </form>
    </body>
    </html>
</jsp:root>
```

8.4. Creating the Second Flow View

The second view in the flow contains one single button that allows the user to exit the flow. The name of the file is “secondView.jspx” and it is to be located next to the first flow view:

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
      <jsp:directive.page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"/>
      <jsp:text>
            <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
      </jsp:text>
      <jsp:text>
            <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
      </jsp:text>

      <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <title>Second View</title>
      </head>
      <body>
        <p>This is the second page in the flow, the Second View.</p>

        <!--
            Clicking the Exit Flow button is to take us out of the flow.
            Nevertheless, it will cause a Spring Web Flow event to be
            sent. This is in order to have all the navigation, be it
            within the flow or to resources outside of the flow, in the
            flow definition file.
        -->
        <form method="post">
            <input type="submit" name="_eventId_toThird" value="Exit Flow"/>
        </form>
      </body>
      </html>
</jsp:root>
```

Note that:

- The Exit Flow button will send a Spring Web Flow event.
  I wanted to have all navigation within the flow and when exited the flow in the flow definition file, so I chose to have the Exit Flow button send an event that, as we will see in the flow definition file, will cause a transition to a flow end state.
  It is also a good practice to have flows transition to an end state when they are finished, in order to allow Spring Web Flow to clean up resources associated with the flow.

8.5. Creating the Service Bean

In this section we implement a service bean that has one method for each type of event in the lifecycle of a flow.

```
/**
 * StartViewService.java
 *
 */
package com.ivan.springwebflowtut.services;

import java.util.Date;

import org.springframework.stereotype.Service;

/**
 * Implements the service backing the start view in the flow flow1.
 * This service contains a number of logging methods that generate console
 * output.
 *
 * @author Ivan A Krizsan
 */
@Service("startViewService")
public class StartViewService
{

    public void enterFlow()
    {
        System.out.println("Flow entered at " + new Date());
    }

    public void exitFlow()
    {
        System.out.println("Flow exited at " + new Date());
    }

    public void viewStateEntered(final String inStateName)
    {
        System.out.println("View state '" + inStateName + "' entered at " +
            new Date());
    }

    public void viewStateExited(final String inStateName)
    {
        System.out.println("View state '" + inStateName + "' exited at " +
            new Date());
    }

    public void transitionPerformed(final String inTransition)
    {
        System.out.println("Transition '" + inTransition + "' performed at " +
            new Date());
    }
}
```

8.6. Creating the Flow Definition File

The flow definition file is named “flow1-webflow.xml” and located in the “flow1” directory. It contains a number of new elements described below.

```
<?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"
    start-state="start">

    <!--
        The <on-start> element allows us to do the following things
        as the flow is entered:
        - Evaluate expressions.
        - Request rendering of one or more fragments.
        - Set the value of an attribute in some scope.
        All the other <on-xxxx> elements in this example allows for
        the same kinds of actions, as listed above.
    -->
    <on-start>
        <evaluate expression="startViewService.enterFlow()"/>
    </on-start>

    <view-state id="start" view="startView.jspx">
        <!--
            The <on-entry> element is similar to the <on-start> element,
            but it enables actions to be invoked etc. when a view state
            is entered.
        -->
        <on-entry>
            <evaluate expression="startViewService.viewStateEntered('start')"/>
        </on-entry>

        <transition on="toSecond" to="second">
            <!--
                A <transition> element can contain <evaluate> expressions,
                allowing expressions to be evaluated when the transition is
                taken.
            -->
            <evaluate expression="startViewService.transitionPerformed('start to
second')"/>
        </transition>

        <!--
            The <on-exit> element is similar to the <on-start> element,
            but it enables actions to be invoked etc. when a view state is exited.
        -->
        <on-exit>
            <evaluate expression="startViewService.viewStateExited('start')"/>
        </on-exit>
    </view-state>

    <view-state id="second" view="secondView.jspx">
        <on-entry>
            <evaluate expression="startViewService.viewStateEntered('second')"/>
        </on-entry>

        <transition on="toThird" to="third">
            <evaluate expression="startViewService.transitionPerformed('second to
third')"/>
        </transition>

        <on-exit>
            <evaluate expression="startViewService.viewStateExited('second')"/>
        </on-exit>
    </view-state>

    <!--
        This is an end state which view does not belong to the flow.
        Such a construct can be used when the flow is to end, but
```

```
        the application is to continue on to another web page, perhaps
        in another flow.
        In the <view-state> and <end-state> elements, the value of the
        view attribute can be prefixed with "externalRedirect:" to indicate
        redirection to web pages outside of the flow.
        Example: externalRedirect:http://www.ivan.com

        In addition, the following additional directives are available:
        serverRelative - Resource relative to the server root.
        contextRelative - Resource relative to the current web application.
        servletRelative - Resource relative to the current servlet.
        Example: externalRedirect:contextRelative:/after-flow.xhtml
        These later directives can also be used when returning a resource
        to redirect to from a FlowHandler.
    -->
    <end-state id="third" view="externalRedirect:contextRelative:/after-flow.xhtml">
        <!--
            An end state may only have an <on-entry> element, but no <on-exit> element.
        -->
        <on-entry>
            <evaluate expression="startViewService.viewStateEntered('third')"/>
        </on-entry>
    </end-state>

    <!--
        The <on-end> element is similar to the <on-start> element,
        but it enables actions to be invoked etc. when a flow ends.
    -->
    <on-end>
        <evaluate expression="startViewService.exitFlow()"/>
    </on-end>
</flow>
```

Note that:

- The <on-start> element allows for actions to be executed etc. as the flow starts.
  In general, all the <on-xxxx> elements allows for the following child elements:
  - <evaluate> - Evaluate expressions.
  - <render> - Request for one or more view fragments to be rendered.
  - <set> - Set the value of an attribute in some scope.
- The <on-entry> element allows for actions to be executed etc. when a view state is entered.
- A <transition> element may contain the following child elements, allowing for actions to be performed in connection to the transition being executed:
  - <evaluate> - Evaluate expressions.
  - <render> - Request for one or more view fragments to be rendered.
  - <set> - Set the value of an attribute in some scope.
- The <on-exit> element allows for actions to be executed etc. when a view state is exited.
- An <end-state> element, denoting an end state of a flow, may contain an <on-entry> element, allowing for actions to be executed etc. when the end state is entered.
- An <end-state> element may not contain an <on-exit> element.
- The value of the *view* attribute of the <end-state> element contains two special prefixes used to redirect to a web page outside of the flow.
  The first prefix, externalRedirect, tells Spring Web Flow that we want to redirect to a web page outside of the flow. The second prefix, contextRelative, indicates that the resource location is given relative to the root of the current web application.
  See comments in the flow definition file for additional prefixes and their functions.
- The <on-end> element allows for actions to be executed etc. when a flow ends.

8.7. Running the Application

If we now run the application and click through it, console output similar to the following will be printed:

```
May 24, 2010 6:25:39 PM org.apache.catalina.core.ApplicationContext log
INFO: Initializing Spring FrameworkServlet 'front-controller'
Flow entered at Mon May 24 18:25:40 CEST 2010
View state 'start' entered at Mon May 24 18:25:40 CEST 2010
Transition 'start to second' performed at Mon May 24 18:25:41 CEST 2010
View state 'start' exited at Mon May 24 18:25:41 CEST 2010
View state 'second' entered at Mon May 24 18:25:41 CEST 2010
Transition 'second to third' performed at Mon May 24 18:25:42 CEST 2010
View state 'second' exited at Mon May 24 18:25:42 CEST 2010
View state 'third' entered at Mon May 24 18:25:42 CEST 2010
Flow exited at Mon May 24 18:25:42 CEST 2010
```

Note that:

- Actions associated to a transition being taken are executed prior to the state from which the transition originates is exited.
- Actions associated with a state having exited are execute before actions associated with having entered a state are executed.
- The figure in the beginning of this chapter lacks a dot for the transition out of the second state and the end state.

This concludes the example showing events associated with different stages of the life-cycle of a flow.

9. Variables and Scopes

In this chapter we will look at parameters entered by a user of the application, variables, different scopes in which variables can exist and, finally, we will see some special variables available to Spring Web Flow applications.
The example program in this chapter will, on the first side of a flow, prompt the user to input his/her name. Upon transition from the first view to the second view, the name input by the user will be saved in multiple variables in different scopes. When arriving on the second page, the contents of the variables in the different scopes will be displayed. Additionally, data obtained by using some special variables is also displayed.
In this example, all expressions are evaluated in the flow definition file and no service beans are used. This is done in order to simplify the example and may not be the best approach when developing real applications.
With the project set up as described in the chapter two, all we need to do is to create two views and a flow definition file.

9.1. Creating the Start View

The first view, the start view, is located in a file named “startView.jspx”, which is located in the “flow1” directory created in the previous section.

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
    <jsp:directive.page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"/>
    <jsp:text>
        <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
    </jsp:text>
    <jsp:text>
        <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
    </jsp:text>

    <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <title>Start View</title>
    </head>
    <body>
        <p>Please enter your name below.</p>

        <form method="post">
            <label for="userName" class="required">User name:</label>
            <input type="text" name="userName"/>
            <br/>
            <br/>
            <input type="submit" name="_eventId_toSecond" value="To Second View"/>
        </form>
    </body>
    </html>
</jsp:root>
```

Note that:

- The value of the *name* attribute of the <input> element is “userName”.
  This also be the name of the parameter containing the name the user of the application entered.

9.2. Creating the Second View

The second view is contained in a file named “secondView.jspx” in the “flow1” directory.

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
      <jsp:directive.page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"/>
      <jsp:text>
            <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
      </jsp:text>
      <jsp:text>
            <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
      </jsp:text>

      <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
          <title>Second View</title>
      </head>
      <body>
        <p>This is the second page in the flow, the Second View.</p>
        <p>
            <h3>Variables from Different Scopes</h3>
            Flow scope: '${flowscope_username}'<br/>
            Flash scope: '${flashscope_username}'<br/>
            Request scope: '${requestscope_username}'<br/>
            View scope: '${viewscope_username}'<br/>
            Conversation scope: '${conversationscope_username}'<br/>
        </p>
        <p>The URL of this view is: ${flowExecutionUrl}</p>
        <p>
            <h3>Message Context</h3>
            Are there any error messages in the MessageContext? ${haserrormsg}<br/>
        </p>
        <p>
            <h3>Flow Execution Context</h3>
            Flow execution has started: ${flowstarted}<br/>
            Flow execution is active: ${flowisactive}<br/>
            Flow execution has ended: ${flowended}
        </p>
        <p>
            <h3>External Context</h3>
            Context path: ${contextpath}<br/>
            Is Ajax request? ${ajaxrequest}<br/>
            Is response allowed? ${responseallowed}<br/>
            Is response complete? ${responsecomplete}<br/>
            Locale: ${currentlocale}
        </p>

      </body>
      </html>
</jsp:root>
```

Note that:

- A number of variables (enclosed by ${}) appear on this page.
  Except for *flowExecutionUrl*, these are all variables assigned values in the flow definition file. We'll see how these variables are assigned in the next section.
- The variable *flowExecutionUrl* is a special variable.
  This variable contains the URL of the current view state, relative to the web application context.

9.3. Creating the Flow Definition File

The flow definition file is a simple one; two view states with one transition from the first to the second view state. However, in addition to that, there are two groups of variable assignments, one in each view state:

- When transitioning from the first view state to the second, the user-entered data is stored in several different variables in different scopes.
- When entering the second view state, several expressions that uses special variables are evaluated and the results are stored in variables. The variables all exist in the view scope.

The flow definition file is named “flow1-webflow.xml” and located in the “flow1” directory.

```
<?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"
    start-state="start">

    <view-state id="start" view="startView.jspx">
        <transition on="toSecond" to="second">
            <!--
                Copy the name entered in the text field by the user to
                variables in different scope.
                In the next view, we'll retrieve the variables and see
                which remain and which are destroyed.
            -->
            <set name="flowScope.flowscope_username"
                value="requestParameters.userName"/>
            <set name="flashScope.flashscope_username"
                value="requestParameters.userName"/>
            <set name="requestScope.requestscope_username"
                value="requestParameters.userName"/>
            <set name="viewScope.viewscope_username"
                value="requestParameters.userName"/>
            <set name="conversationScope.conversationscope_username"
                value="requestParameters.userName"/>
        </transition>
    </view-state>

    <!--
        If this state would be an end state, then we would not be able to
        store any values in the view scope, since the flow would be destroyed.
    -->
    <view-state id="second" view="secondView.jspx">
        <!--
            These expressions cannot be evaluated in the JSP, so we evaluate
            them here and store the results in variables in the view scope.
        -->
        <on-entry>
            <!-- Accessing the Spring Web Flow message context. -->
            <evaluate expression="messageContext.hasErrorMessages()"
                result="viewScope.haserrormsg"/>

            <!-- Accessing the Spring Web Flow flow execution context. -->
            <evaluate expression="flowExecutionContext.hasStarted()"
                result="viewScope.flowstarted"/>
            <evaluate expression="flowExecutionContext.isActive()"
                result="viewScope.flowisactive"/>
            <evaluate expression="flowExecutionContext.hasEnded()"
                result="viewScope.flowended"/>

            <!-- Accessing the Spring Web Flow external context. -->
            <evaluate expression="externalContext.getContextPath()"
                result="viewScope.contextpath"/>
            <evaluate expression="externalContext.isAjaxRequest()"
                result="viewScope.ajaxrequest"/>
            <evaluate expression="externalContext.isResponseAllowed()"
```

```
                result="viewScope.responseallowed"/>
            <evaluate expression="externalContext.isResponseComplete()"
                result="viewScope.responsecomplete"/>

            <!--
                Note that when retrieving the locale, the getter method
                is not used. Instead we just write the name of the
                property.
            -->
            <evaluate expression="externalContext.locale"
                result="viewScope.currentlocale"/>
        </on-entry>
    </view-state>
</flow>
```

Note that:

- To set the value of variables when the transition from the first view state to the second view state is taken, a number of <set> elements are used in the <transition> element.
- To set the value of a variable, the <set> element is used with the attributes *name* and *value*. The *name* attribute specifies the scope and name of the variable to be assigned in the form "scopeName.variableName". The *value* attribute specifies the value to assign to the variable. The next section lists the different scopes in Spring Web Flow.
- All the variables are assigned the value specified by "requestParameters.userName". The *requestParameters*-part is a special variable that can be used to access client request parameters, that is, parameters entered by the user. The *userName*-part is the name we gave the <input> element in the first view.
- The second view state is a view state and not an end state.
  I first made the mistake to make this state an end state, but this caused problems since the flow was destroyed upon entering this stage.
- When entering the second view state, a number of expressions are evaluated and the results are placed in different variables in the view scope.
  All of these expressions contains special variables allowing us to access different contexts. A subsequent section lists the special EL variables available in Spring Web Flow.

9.4. Spring Web Flow Scopes

The following special EL variables are available to access the different scopes available in a Spring Web Flow flow:

Scope	Scope Life-Span
flowScope	Exists for the entire life of the flow.
flashScope	Exists for the entire life of the flow and is cleared after every view render.
requestScope	Exists from the time a flow is called, as a result of a user request, to the time when it returns.
viewScope	Exists from when a view state is entered to when it is exited.
conversationScope	Exists during the lifetime of a top-flow and all its sub-flows. Shared by the top-flow and all its sub-flows.

When accessing the contents of a variable, the scope may be specified. If the scope is not specified, the above scopes will be searched in the following order:

1. Request scope.
2. Flash scope.
3. View scope.
4. Flow scope.
5. Conversation scope.

Note that in an end state, the flow ceases to exist and thus most of the above scopes that are associated with the flow. If we want to render a view and display some data in connection to the end state, it is still possible to place the data in the request scope.

9.5. Spring Web Flow Special Variables

Apart from the special variables used to access the different contexts, the following special EL variables are available in a Spring Web Flow flow:

Special Variable	Type	Description
requestParameters	Map(?)	Contains client request parameters.
currentEvent	org.springframework.webflow.execution.Event	Object representing the current event, in which attributes may be stored.
currentUser	java.security.Principal	User's authenticated *Principal*.
messageContext	org.springframework.binding.message.MessageContext	Context object in which messages can be added and retrieved.
resourceBundle	?	Used to access messages in a resource bundle.
flowRequestContext	org.springframework.webflow.execution.RequestContext	Context holding information about the current request.
flowExecutionContext	org.springframework.webflow.execution.FlowExecutionContext	Context containing information about the execution of the current flow.
flowExecutionUrl	String	URL of the current view state in the flow, relative to the web application context.
externalContext	org.springframework.webflow.context.ExternalContext	Represents the context of the current client request made that is to manipulate the execution of the flow.

9.6. Running the Application

If we now run the example application and enter the name “Ivan” on the first page, the second page will look something like this:

```
This is the second page in the flow, the Second View.

Variables from Different Scopes
Flow scope: 'Ivan'
Flash scope: 'Ivan'
Request scope: ''
View scope: ''
Conversation scope: 'Ivan'

The URL of this view is: /SpringWebFlowTutorial2/flow/flow1?execution=e1s3

Message Context
Are there any error messages in the MessageContext? false

Flow Execution Context
Flow execution has started: true
Flow execution is active: true
Flow execution has ended: false

External Context
Context path: /SpringWebFlowTutorial2
Is Ajax request? false
Is response allowed? true
Is response complete? false
Locale: en_US
```

Note that:

- The contents of the flow, flash and conversation scope variables set when transitioning from the first view state was preserved.
- The contents of the request and view scope variables set when transitioning from the first view state was lost.
- The values from the different special variables are displayed.
  This includes the the current locale from the external context.

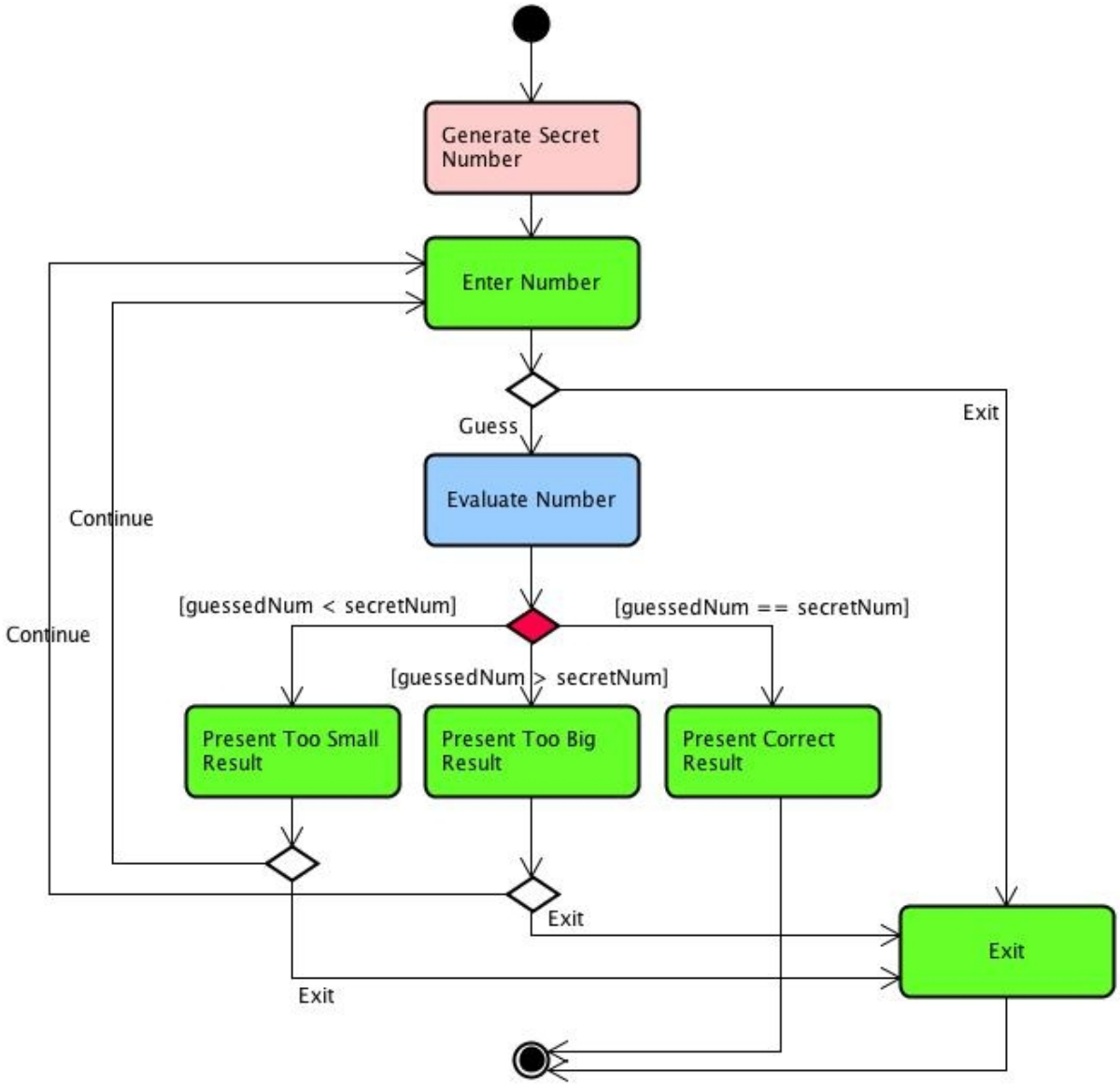
This concludes the example program on variables and scopes.

10. Action and Decision States

In this chapter we will write a game, a web-based number-guessing game. This will enable us to learn how to use action and decision states. In addition, we will also see how to design a web application using Spring Web Flow that requires the application to keep some kind of information, state, for each user. In the example application, the state kept consists of the secret number that is the correct answer and the last number guessed by the user.
The flow developed in this chapter will also be thoroughly tested, using knowledge from the example in chapter five. We will also see some new aspects of unit testing flows, such as mocking of service beans.

10.1. Analyzing the Flow

In the interest of developing a correct application, we start by analyzing the flow of the number-guessing game. The following activity diagram that describes the flow of the number-guessing game:



Activity diagram describing the flow in the number-guessing game.

The following colours have been used to denote different types of nodes:

- Green denotes a view state node.
- Red denotes a decision state node.
- Pink denotes an action performed when the flow starts.
- Light-blue denotes an action state node.

Note that there are different ways of analyzing the number-guessing game. The above diagram describes one possible solution.

Later, prior to creating the flow definition file, we will use the above activity diagram to create a unit test for the flow.

With the project set up as described in the chapter two, we start by creating the different views.

10.2. Creating the Enter Guess View

The enter guess view is the view in which the user can choose to enter a new guess (number) or to exit the game. This view is located in the "flow1" directory, with the name "enterNumber.jspx".

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root
    xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    version="2.0">
    <jsp:directive.page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"/>
    <jsp:text>
        <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
    </jsp:text>
    <jsp:text>
        <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
    </jsp:text>

    <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <title>Guess a Number</title>
    </head>
    <body>

        <p>
            <c:choose>
                <c:when test="${empty lastGuessedNumber}">
                    There is no last guessed number yet.
                </c:when>
                <c:otherwise>
                    The last guessed number is: ${lastGuessedNumber}
                </c:otherwise>
            </c:choose>
        </p>

        <p>Please guess a new number below.</p>
        <form method="post">
            <label for="newGuessedNumber" class="required">Guessed number:</label>
            <input type="text" name="newGuessedNumber"/>
            <br/>
            <br/>
            <input type="submit" name="_eventId_setNewGuessedNumber" value="Submit"/>
            <input type="submit" name="_eventId_toExit" value="Exit"/>
        </form>
    </body>
    </html>
</jsp:root>
```

10.3. Creating the Correct Guess View

This view tells the user that he/she has guessed the secret number.
The file is named “correctGuess.jspx” and is, as usual, located in the “flow1” directory.

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
    <jsp:directive.page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"/>
    <jsp:text>
        <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
    </jsp:text>
    <jsp:text>
        <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
    </jsp:text>

    <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <title>Correct!</title>
    </head>
    <body>
        <h3>You guessed the correct number!</h3>
        <p>The correct number is ${lastGuessedNumber}. Congratulations!</p>
    </body>
    </html>
</jsp:root>
```

10.4. Creating the Exited Game View

This view is displayed when the user chooses to exit the game.
The file is named “exitedGame.jspx” and is located in the “flow1” directory.

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
    <jsp:directive.page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"/>
    <jsp:text>
        <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
    </jsp:text>
    <jsp:text>
        <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
    </jsp:text>

    <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <title>Exited Game</title>
    </head>
    <body>
        <h3>You have exited the game.</h3>
        <p>The secret number was: ${secretNumber}</p>
    </body>
    </html>
</jsp:root>
```

10.5. Creating the Too High/Low Views

Next, we create the views telling the user that his/her guess was too high or too low. For the two web pages, there are all in all three files; one containing a common section, one containing the too-high page and one containing the too-low page.

Creating the Common Segment

The common part of the two pages shows the last number guessed and asks the user about what action to take next.
It is located in the “flow1” directory, in a file named “guessResultCommon.jsp”.

```
<p>You guessed: ${lastGuessedNumber}</p>

<form method="post">
    <input type="submit" name="_eventId_toStart" value="Enter Another Number"/>
    <input type="submit" name="_eventId_toExit" value="End"/>
</form>
```

Creating the Too High View

The too-high view is shown when the guessed number is larger than the secret number. The name of the file is “tooHigh.jspx” and it is also located in the “flow1” directory.

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
    <jsp:directive.page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"/>
    <jsp:text>
        <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
    </jsp:text>
    <jsp:text>
        <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
    </jsp:text>

    <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <title>Too High</title>
    </head>
    <body>
        <h3>Your guess is too high!</h3>

        <jsp:include page="guessResultCommon.jsp"/>
    </body>
    </html>
</jsp:root>
```

Creating the Too Low View

Not surprisingly, the too-low view is shown when the number entered by the user is less than the secret number.
The filename is “tooLow.jspx” and the file is located in the “flow1” directory.

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
    <jsp:directive.page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"/>
    <jsp:text>
        <![CDATA[ <?xml version="1.0" encoding="UTF-8" ?> ]]>
    </jsp:text>
    <jsp:text>
        <![CDATA[ <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ]]>
    </jsp:text>

    <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <title>Too Low</title>
    </head>
    <body>
        <h3>Your guess is too low!</h3>

        <jsp:include page="guessResultCommon.jsp"/>
    </body>
    </html>
</jsp:root>
```

10.6. Creating the Service Interface and Bean

The service bean of the game has two responsibilities; it creates a secret number for each game played and it determines if a guess is correct or not.
If we consider that our game may have more than one single user, we should be aware that there will be only one single instance of the service bean. This means that:

- The service bean must be stateless.
  No data with relation to a specific user or a specific game can be stored in the service, since the service will serve multiple games.
- The service bean must be threadsafe.
  The service bean is a plain Java object (POJO) and, with multiple players, it may be called by many threads simultaneously.

Of course it is possible to configure the service bean as having the bean-scope “prototype” and then retrieve a new instance of the service at the start of each game and use it throughout the game.
As far as I see it, it is a question of taste – if you need a stateful service, then use the “prototype” bean-scope and store an instance of the service in the Spring Web Flow's conversation or flow scope.
I have chosen to implement a stateless service and put the state in the conversation scope of the flow, since I feel it is the simplest solution.

In order to make mocking the service easier, we first implement a service interface:

```
package com.ivan.springwebflowtut.services;

/**
 * Defines the properties of the guess number service in the
 * number guessing game.
 *
 * @author Ivan A Krizsan
 */
public interface GuessNumberService
{

    /**
     * Generates a random secret number within a certain range.
     */
    public abstract long generateSecretNumber();

    /**
     * Determines if the supplied number is greater than, less than
     * or equal to the supplied secret number.
     *
     * @param inGuessedNumber Number to compare to the secret number.
     * @param inSecretNumber The secret number.
     * @return Zero if the guessed number is equal to the secret number,
     * a positive number if the guessed number is greater than the
     * secret number and a negative number if the guessed number is less
     * than the secret number.
     */
    public abstract int isEqualToSecretNumber(final long inGuessedNumber,
        final long inSecretNumber);
}
```

The service bean is then implemented as follows:

```
package com.ivan.springwebflowtut.services.impl;

import org.springframework.stereotype.Service;
import com.ivan.springwebflowtut.services.GuessNumberService;

/**
 * Implements the service for the Guess-the-Number game.
 * This service, used by a Spring Web Flow application, will exist
 * in one single instance that is used by all the users of the application.
 * Thus the service must be stateless and all the methods must also be
 * thread-safe.
 *
 * @author Ivan A Krizsan
 */
@Service("guessNumberService")
public class GuessNumberServiceImpl implements GuessNumberService
{
    /* Constant(s): */
    private final static long SECRET_NUMBER_RANGE = 100;

    @Override
    public long generateSecretNumber()
    {
        long theSecretNumber = (long)(Math.random() * SECRET_NUMBER_RANGE);
        System.out.println("CurrentNumberService.generateSecretNumber()");

        return theSecretNumber;
    }

    @Override
    public int isEqualToSecretNumber(final long inGuessedNumber,
        final long inSecretNumber)
    {
        int theCompareResult = new Long(inGuessedNumber).compareTo(inSecretNumber);

        System.out.println("CurrentNumberService.isEqualToSecretNumber() - "
            + "Guessed value: " + inGuessedNumber + ", compare result: "
            + theCompareResult);

        return theCompareResult;
    }
}
```

10.7. Creating the Flow Unit Test

Using the activity diagram created earlier, we can now, before creating the flow definition file, implement a unit test for the flow of the number-guessing game. In "src/test/java", in the appropriate package, create the following unit test for the flow:

```
package com.ivan.springwebflowtut;

import junit.framework.Assert;

import org.springframework.webflow.config.FlowDefinitionResource;
import org.springframework.webflow.config.FlowDefinitionResourceFactory;
import org.springframework.webflow.core.collection.LocalAttributeMap;
import org.springframework.webflow.core.collection.MutableAttributeMap;
import org.springframework.webflow.test.MockExternalContext;
import org.springframework.webflow.test.MockFlowBuilderContext;
import org.springframework.webflow.test.execution.AbstractXmlFlowExecutionTests;

import com.ivan.springwebflowtut.services.GuessNumberService;

/**
 * This class implements tests of the "flow1" Spring Web Flow flow.
 *
 * @author Ivan A Krizsan
 */
public class Flow1Test extends AbstractXmlFlowExecutionTests
{
    /* Constant(s): */
    private final static String FLOW1_FILE_LOCATION =
        "src/main/webapp/WEB-INF/flows/flow1/flow1-webflow.xml";
    private final static long SECRET_NUMBER = 49L;
    private final static String GUESSNUMBER_SERVICE_BEANNAME =
        "guessNumberService";

    /* Flow states. */
    private final static String ENTER_NUMBER_STATE = "enterNumber";
    private final static String TOO_SMALL_STATE = "guessTooSmall";
    private final static String TOO_BIG_STATE = "guessTooBig";

    /* Flow events. */
    private final static String EXIT_EVENT = "toExit";
    private final static String SET_GUESS_EVENT = "setNewGuessedNumber";
    private final static String TO_START_EVENT = "toStart";

    /* Flow parameters. */
    private final static String GUESSED_NUMBER_PARAM = "newGuessedNumber";

    /* Instance variable(s): */
    private MutableAttributeMap mFlowInputParams;
    private MockExternalContext mMockExternalContext;

    /**
     * Retrieves the flow resource for the flow to test using the
     * supplied resource factory.
     *
     * @param inResourceFactory Factory used to create flow resource.
     * @return Flow resource of flow to test.
     */
    @Override
    protected FlowDefinitionResource getResource(
        final FlowDefinitionResourceFactory inResourceFactory)
    {
        FlowDefinitionResource theFlowResource;

        theFlowResource =
            inResourceFactory.createFileResource(FLOW1_FILE_LOCATION);
        return theFlowResource;
    }

    /**
     * Creates flow parameters map and mock context used when testing
     * a flow.
     */
    private void createParamsMapAndMockContext()
```

```
    {
        mFlowInputParams = new LocalAttributeMap();
        mMockExternalContext = new MockExternalContext();
    }

    /**
     * Configures the builder context for the flow being tested.<br/>
     * Here we can create mocks for service beans and subflows that
     * the flow we are testing uses.
     *
     * @param inBuilderContext Builder context to be configured.
     */
    @Override
    protected void configureFlowBuilderContext(
        MockFlowBuilderContext inBuilderContext)
    {
        GuessNumberService theMockService;

        /* Create a mock guess-number service. */
        theMockService = new GuessNumberService()
        {

            @Override
            public long generateSecretNumber()
            {
                return SECRET_NUMBER;
            }

            @Override
            public int isEqualToSecretNumber(long inGuessedNumber,
                long inSecretNumber)
            {
                /*
                 * Make sure that the flow sends us the secret number
                 * generated by the service.
                 */
                Assert.assertEquals(SECRET_NUMBER, inSecretNumber);

                Long theGuessedNumber = inGuessedNumber;
                return theGuessedNumber.compareTo(inSecretNumber);
            }

        };

        /*
         * Register the mock service using the bean name so that
         * Spring Web Flow will use the mock when running the unit
         * test.
         */
        inBuilderContext.registerBean(GUESSNUMBER_SERVICE_BEANNAME,
            theMockService);
    }

    /**
     * Tests starting the flow and the initial state of the flow.
     */
    public void testFlowStartup()
    {
        createParamsMapAndMockContext();
        startFlow(mFlowInputParams, mMockExternalContext);
        assertCurrentStateEquals(ENTER_NUMBER_STATE);
    }

    /**
     * Tests sending the flow an exit event when it is in the
     * enter-number state.
     */
    public void testExitFromEnterNumber()
    {
        createParamsMapAndMockContext();

        setCurrentState(ENTER_NUMBER_STATE);
        mMockExternalContext.setEventId(EXIT_EVENT);

        resumeFlow(mMockExternalContext);
        assertFlowExecutionEnded();
    }
```

```
    /**
     * Tests making a too small guess.
     */
    public void testTooSmallGuess()
    {
        createParamsMapAndMockContext();

        startFlow(mFlowInputParams, mMockExternalContext);

        mMockExternalContext.setEventId(SET_GUESS_EVENT);

        /* Use the mock external context to set request parameters. */
        mMockExternalContext.putRequestParameter(GUESSED_NUMBER_PARAM, ""
            + (SECRET_NUMBER - 1));

        resumeFlow(mMockExternalContext);
        assertCurrentStateEquals(TOO_SMALL_STATE);
    }

    /**
     * Tests making a too big guess.
     */
    public void testTooBigGuess()
    {
        createParamsMapAndMockContext();

        startFlow(mFlowInputParams, mMockExternalContext);

        mMockExternalContext.setEventId(SET_GUESS_EVENT);

        /* Use the mock external context to set request parameters. */
        mMockExternalContext.putRequestParameter(GUESSED_NUMBER_PARAM, ""
            + (SECRET_NUMBER + 1));

        resumeFlow(mMockExternalContext);
        assertCurrentStateEquals(TOO_BIG_STATE);
    }

    /**
     * Tests making a correct guess.
     */
    public void testCorrectGuess()
    {
        createParamsMapAndMockContext();

        startFlow(mFlowInputParams, mMockExternalContext);

        mMockExternalContext.setEventId(SET_GUESS_EVENT);

        /* Use the mock external context to set request parameters. */
        mMockExternalContext.putRequestParameter(GUESSED_NUMBER_PARAM, ""
            + SECRET_NUMBER);

        resumeFlow(mMockExternalContext);
        assertFlowExecutionEnded();
    }

    /**
     * Tests sending the flow an exit event when it is in the
     * too-small state.
     */
    public void testExitFromTooSmall()
    {
        createParamsMapAndMockContext();

        setCurrentState(TOO_SMALL_STATE);
        mMockExternalContext.setEventId(EXIT_EVENT);

        resumeFlow(mMockExternalContext);
        assertFlowExecutionEnded();
    }

    /**
     * Tests sending the flow an exit event when it is in the too-big
     * state.
     */
```

```
    public void testExitFromTooBig()
    {
        createParamsMapAndMockContext();

        setCurrentState(TOO_BIG_STATE);
        mMockExternalContext.setEventId(EXIT_EVENT);

        resumeFlow(mMockExternalContext);
        assertFlowExecutionEnded();
    }

    /**
     * Tests sending the flow a continue event when it is in the
     * too-small state.
     */
    public void testContinueFromTooSmall()
    {
        createParamsMapAndMockContext();

        setCurrentState(TOO_SMALL_STATE);
        mMockExternalContext.setEventId(TO_START_EVENT);

        resumeFlow(mMockExternalContext);
        assertCurrentStateEquals(ENTER_NUMBER_STATE);
    }

    /**
     * Tests sending the flow a continue event when it is in the
     * too-big state.
     */
    public void testContinueFromTooBig()
    {
        createParamsMapAndMockContext();

        setCurrentState(TOO_BIG_STATE);
        mMockExternalContext.setEventId(TO_START_EVENT);

        resumeFlow(mMockExternalContext);
        assertCurrentStateEquals(ENTER_NUMBER_STATE);
    }
}
```

Note that:

- In the *configureFlowBuilderContext* method, a mock for the guess-number service that will be used from the flow is created and registered.
- To simulate parameters entered by users in a form, use the *putRequestParameter* method in the *MockExternalContext* class.
  An example of how such a parameter is entered can be seen, for instance, in the *testTooSmallGuess* method.

If we run the above unit test now, it fails, of course.

10.8. Creating the Flow Definition File

With all the above pieces in place, the only thing missing is the flow definition file. This is where action and decision states, the main topics of this chapter, are used.
The flow definition file is named “flow1-webflow.xml” and located in the “flow1” directory.

```
<?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"
    start-state="enterNumber">

    <!--
        When the flow starts, ask the service to generate a secret number.
    -->
    <on-start>
        <evaluate
            expression="guessNumberService.generateSecretNumber()"
            result="conversationScope.secretNumber"/>
    </on-start>

    <!--
        The view in which the user enters a number.
    -->
    <view-state id="enterNumber" view="enterNumber.jspx">
        <!--
            Slightly contrieved for this example:
            Having received a number, transition to a special state,
            an action state, that evaluates the number.
        -->
        <transition on="setNewGuessedNumber" to="evaluateNewGuess">
            <!--
                Put the number entered by the user into a variable in
                the conversation scope so that it is available to
                subsequent views.
                Note that we tell Spring Web Flow what kind of value is
                expected.
            -->
            <set name="conversationScope.lastGuessedNumber"
                value="requestParameters.newGuessedNumber" type="long"/>
        </transition>
      </view-state>

    <!--
        An action state is used to evaluate one or more expressions.
        It is also possible to, depending on the result of an evaluation,
        select which subsequent state to transition to.
        Again, contrieved for this example, this state only evaluates an
        expression, stores the result in a flash scope variable and then
        transitions on to the next state in which decisions are taken.
    -->
    <action-state id="evaluateNewGuess">
        <evaluate
            expression="guessNumberService.isEqualToSecretNumber(conversationScope.lastG
uessedNumber, conversationScope.secretNumber)"
            result="flashScope.guessResult">
        </evaluate>
        <transition to="chooseGuessResult"/>
    </action-state>

    <!--
        A decision state that, using the result from a previous state,
        determines which view to show next.
    -->
    <decision-state id="chooseGuessResult">
        <if test="guessResult &lt; 0" then="guessTooSmall"/>
        <if test="guessResult > 0" then="guessTooBig"/>
        <if test="guessResult == 0" then="guessCorrect"/>
    </decision-state>

    <view-state id="guessTooSmall" view="tooLow.jspx">
```

```
    </view-state>

    <view-state id="guessTooBig" view="tooHigh.jspx">
    </view-state>

    <end-state id="guessCorrect" view="correctGuess.jspx">
    </end-state>

    <end-state id="exit" view="exitedGame.jspx">
    </end-state>

    <!--
        Almost all views in the flow allows the user either to end the
        game or to go to the view in which to enter a guess.
        Thus these transitions are placed in the following element,
        reducing duplication in the view states.
    -->
    <global-transitions>
        <transition on="toStart" to="enterNumber"/>
        <transition on="toExit" to="exit"/>
    </global-transitions>
</flow>
```

Note that:

- When the flow starts, we ask the service bean implemented in the previous section to create a secret number for the game.
  The secret number is stored in the Spring Web Flow's conversation scope for the duration of the game.
- In the “enterNumber” view state, the flow transitions to a special state, an action state, to evaluate the new guess.
- Prior to evaluating the new guess, the number is copied from the request parameter to a variable in conversation scope.
  This is to preserve the last guess throughout the flow, until a new guess is made.
- In the <set> element copying the request parameter, we use the type attribute with the value “long” to tell Spring Web Flow that we expect a long integer.
  This does not mean that it is safe for users to enter non-numeric data. If the user does that, a *NumberFormatException* will be thrown.
- The <action-state> element specifies a state that is used to evaluate one or more expressions. In the example, the service is asked to determine whether the guessed number is equal to, larger than or less than the secret number.
  The result is placed in a flash scope variable, which will be available to the next state since this state is an action state. Recall that flash scoped variables are cleared after every view rendering.
- It is possible to take decisions in an action state.
  In such a case, an expression is evaluated without having to place the result in a variable. One or more <transition> elements are then used, with the value(s) of the *on* attributes matching the possible results from the evaluated expression.
- In the action state there is a transition that is always taken.
  Such a transition does not use the *on* attribute, only the *to* attribute.
- The <decision-state> element examines the result from the action state, stored in the flash scope variable guessResult, and determines which view to transition to.
- The <decision-state> element can contain any number of <if> elements.
  An <if> element contains a *test* attribute, which contains an expression that evaluates to

boolean true or false, and a *then* attribute that contains a state id to which the decision state will transit to if the expression in the *test* attribute evaluated to true.
Optionally, the <if> element can also contain an *else* attribute, specifying a state id to which to transfer to if the test expression evaluated false.

- The flow has two end states; *guessCorrect* and *exit*.
- In order to reduce duplication and complexity, two transitions that are common to almost all the states in the flow have been refactored out to a <global-transitions> element.

10.9. Running the Flow Unit Test

If we now run the flow unit test implemented earlier, it should pass all tests. The flow unit test can come in handy if we want to modify the flow definition file, like in the next section.

10.10. Decisions in Action States

Out of curiosity I rewrote the decision state in the above flow using action states. The result is as follows:

```
...
    <action-state id="chooseGuessResult">
        <evaluate expression="guessResult &lt; 0"/>
        <transition on="yes" to="guessTooSmall"/>
        <transition to="chooseGuessResult2"/>
    </action-state>

    <action-state id="chooseGuessResult2">
        <evaluate expression="guessResult > 0"/>
        <transition on="yes" to="guessTooBig"/>
        <transition to="guessCorrect"/>
    </action-state>
...
```

Note that:

- Using action states with conditions that have more than two possibilities quickly becomes verbose.
- Directly examining the *guessResult* variable and transitioning on the different values it can have (-1, 0, 1) would have allowed me to avoid the second <action-state> element.
- The flow unit test still passes all tests.

10.11. Running the Application

If we now run the example application and enter numbers until we guess the correct number, the application should end with a webpage telling you that you guessed the correct number.
There should also be console output similar to this:

```
CurrentNumberService.generateSecretNumber()
CurrentNumberService.isEqualToSecretNumber() - Guessed value: 50, compare result: -1
CurrentNumberService.isEqualToSecretNumber() - Guessed value: 80, compare result: 1
CurrentNumberService.isEqualToSecretNumber() - Guessed value: 65, compare result: 1
CurrentNumberService.isEqualToSecretNumber() - Guessed value: 55, compare result: -1
CurrentNumberService.isEqualToSecretNumber() - Guessed value: 57, compare result: -1
CurrentNumberService.isEqualToSecretNumber() - Guessed value: 60, compare result: -1
CurrentNumberService.isEqualToSecretNumber() - Guessed value: 63, compare result: -1
CurrentNumberService.isEqualToSecretNumber() - Guessed value: 64, compare result: 0
```

Note that:

- First, a secret number is generated.
- There are a number of calls to the *isEqualToSecretNumber* method.
  Each of these calls represent a guess from the user and we can see the guessed value, as well as the result when the guessed value was compared to the secret number.

If we then:

- Clear all logs.
- Start the example application again by navigating to http://localhost:8080/SpringWebFlowTutorial2
- Entering two guesses.
- Open a new browser window or tab and navigate to http://localhost:8080/SpringWebFlowTutorial2 again.
- Enter two more guesses in the the new window/tab.

The above sequence simulates two different users using our application. If we look at the console log, we can see something similar to:

```
CurrentNumberService.generateSecretNumber()
CurrentNumberService.isEqualToSecretNumber() - Guessed value: 34, compare result: -1
CurrentNumberService.isEqualToSecretNumber() - Guessed value: 80, compare result: 1
CurrentNumberService.generateSecretNumber()
CurrentNumberService.isEqualToSecretNumber() - Guessed value: 34, compare result: -1
CurrentNumberService.isEqualToSecretNumber() - Guessed value: 80, compare result: -1
```

Note that:

- The generateSecretNumber method is called twice.
  The second time, it is called after two guesses have been entered.
- The compare result differs, in the case of the above log, when the number 80 is entered.
- Clicking the Exit or End button, depending on which page is visible, for both the users will produce two different secret numbers. This was expected and indicates that our application works correctly.

This concludes the example that, among other things, shows the use of action- and decision-states in Spring Web Flow.

11. Flow Inheritance

In this chapter we will take a look at an example showing flow- and flow-state inheritance. This is a useful feature if you want to specify common properties of flows in one single place and then let a number of flows incorporate such properties. A concrete example is error handling – it is commonly identical for all the flows in an application.

Spring Web Flow supports these two types of inheritance:

- Flow inheritance.
  A flow can inherit from one or more flows.
- Flow state inheritance.
  A state in a flow can inherit from one single state of the same type in a parent flow.

In addition to flow inheritance, it is also possible to make a flow abstract. An abstract flow is designed to be a parent of other flows and cannot be run.

11.1. Setting up the Example Project

As a starting point for this chapter's example program, we will use the example from chapter three. Before continuing, set up a new project as described in chapter two and make all the additions described in chapter three. Make sure that the application works correctly.

We are now ready to refactor out part of the behaviour of the original flow to a parent flow.

11.2. Creating the Parent Flow Definition File

The parent flow will, as recommended by best practices, reside in a directory of its own.

- In the “flows” directory in the project's WEB-INF directory, create a new directory named “parentflow1”. The parent flow will be assigned the name of this directory as its id.
- In the “parentflow1” directory, create a flow definition file with the name “parent1-webflow.xml”, with the following contents:

```
<?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"
   abstract="true">
   <!--
        Note that the abstract attribute of the <flow> element is
        set to true.
    -->

    <!--
        Specify a number of transitions available from anywhere in
        a flow.
    -->
    <global-transitions>
        <transition on="toStart" to="start"/>
        <transition on="toSecond" to="second"/>
        <transition on="toThird" to="third"/>
    </global-transitions>
</flow>
```

11.3. Flow Inheritance

With a parent flow in place, we can now refactor the original flow in the file "flow1-webflow.xml" according to the following steps:

- In the <flow> element, add a *parent* attribute with the value being the id of the parent flow: `parent="parentflow1"`
- Remove all the transitions in the two view states.

The resulting flow definition file looks like this (old comments have been removed):

```
<?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"
   start-state="start"
   parent="parentflow1">
   <!--
        Note that the id of the parent flow is specified using the
        parent attribute in the <flow> element.
    -->

    <view-state id="start" view="startView.jspx">
    </view-state>

    <view-state id="second" view="secondView.jspx">
    </view-state>

    <end-state id="third" view="thirdView.jspx"/>
</flow>
```

11.4. Running the Application

If we now run the application, it should work as before. This is a time when a flow unit test would have come in handy – it would have been able to quickly verify whether the refactored flow definition is correct or not.

11.5. Creating the Parent State Flow Definition File

In this step we will create the flow definition file that contains a view state from which we will inherit. Again, according to best practices, the flow definition file is to reside in a directory of its own. This is also required in order for the flow to have an unique id.

- In the “flows” directory in the project's WEB-INF directory, create a new directory named “parentviewflow1”. The parent flow will be assigned the name of this directory as its id.
- In the “parentviewflow1” directory, create a flow definition file with the name “parent2-webflow.xml”, with the following contents:

```
<?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"
    abstract="true">

    <!--
        This is the parent view state that specifies a view that will
        be rendered when the state is entered.
    -->
      <view-state id="parent-viewstate" view="startView.jspx">
      </view-state>
</flow>
```

Note that:

- The view state with the id “parent-viewstate” specifies that the view rendered when the view state is entered is in the file “startView.jspx”.
- Again, the flow is defined to be abstract.

11.6. (View) State Inheritance

With the parent flow in place, we can now refactor the original flow in the file “flow1-webflow.xml” according to the following steps:

- In the <view-state> element with the id “start”, remove the *view* attribute and the associated value.
- In the <view-state> element with the id “start”, add a *parent* attribute with the value “parentviewflow1#parent-viewstate”.
  `parent="parentviewflow1#parent-viewstate"`

The resulting flow definition file looks like this:

```
<?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"
   start-state="start"
   parent="parentflow1">
   <!--
        Note that the id of the parent flow is specified using the
        parent attribute in the <flow> element.
    -->

    <!--
        This view state inherits from the view state with the name
        "parent-viewstate" in the parent flow "parentviewflow1".
        A state can only inherit from one single state of the same
        type, that is, a view state can only inherit from another
        view state.
    -->
    <view-state id="start" parent="parentviewflow1#parent-viewstate">
    </view-state>

    <view-state id="second" view="secondView.jspx">
    </view-state>

    <end-state id="third" view="thirdView.jspx"/>
</flow>
```

Note that:

- The view state “start” inherits from the view state “parent-viewstate” in the flow “parentviewflow1”.
- A view can only inherit from one single view.

11.7. Running the Application

If we now run the application, the first page should appear as it did prior to the last refactoring. Again, this is a time when a flow unit test would have come in handy.

This concludes the example on flow inheritance and this chapter.

12. Developing Spring Web Flow Applications

This chapter contains some advice concerning developing applications that uses Spring Web Flow. These advice are based on experiences I have made and are quite subjective - that is, they may or may not be suitable in different circumstances. If you have constructive critique or other suggestions in regard to this topic, I would very much like to hear from you.

12.1. Structural Recommendations

The validity of the following structural recommendations depend to a large extent on the other technologies used in connection to Spring Web Flow. If you for instance use Spring MVC, then you may want to skip this section altogether, since your application will already be structured according to the MVC design pattern.

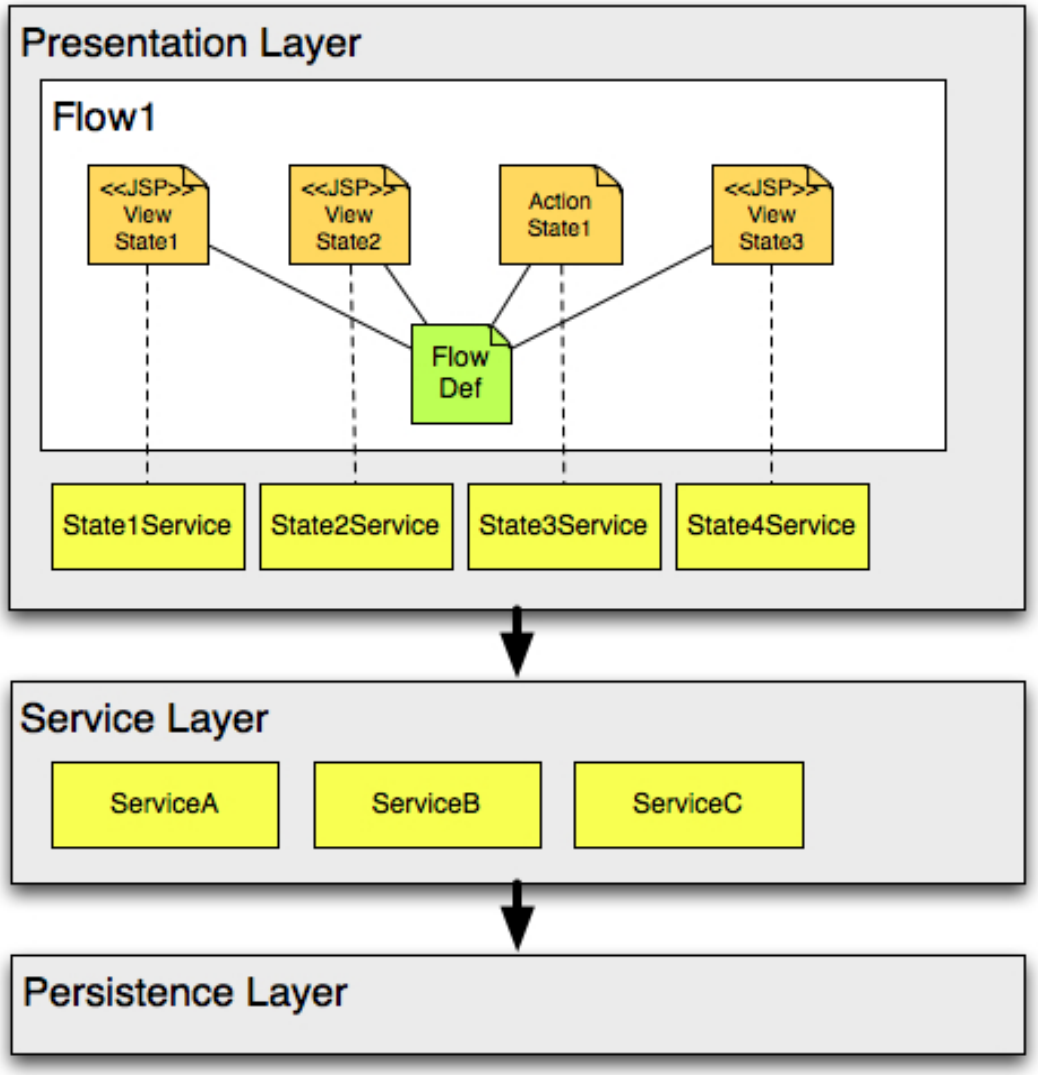
The basic ideas of the structures proposed here are:

- Introduce one or more services in the presentation layer.
  These services are implemented as Spring beans and may contain dependencies on Spring Web Flow and/or the presentation technology used.
  The main responsibilities of such services are to delegate work to services in the service layer below the presentation layer and to adapt the data obtained from these service so that it is suitable for presentation.
  The services in the presentation layer also isolates the service layer preventing any direct dependencies from flow definition files and presentation technology specific files, like xhtml files and JSPs, on the services in the service layer. This enables convenient refactoring of the service layer, with full support from IDE tooling and the compiler.
- Group the services, flows and views.
  A flow, its views and the presentation layer services should be grouped in a consistent manner that makes it easy to predict and navigate.
  With one service in the presentation layer per state (view state etc), or per flow definition, I can immediately navigate to the corresponding service, without having to look at the flow definition and/or the file containing the presentation (for instance the JSP file, the xhtml file etc.).
  Additionally, with a fixed structure, all developers on a project will more easier be able to extend and maintain code written by others.
- Generalize the service layer.
  The service layer can be made more general which increase the chance of being able to expose it, for instance, using some kind of web service technology, to third party.
  It also increases the ability to reuse services when the application is extended.

The two strategies below may be combined within one and the same application, as seen fit.

One Presentation Service Per State

The first suggestion is more appropriate for flows that contain a larger number of states. Each state, be it view states, action states etc, in a Spring Web Flow flow has a dedicated service bean in the presentation layer. States may only invoke methods on their corresponding service bean.

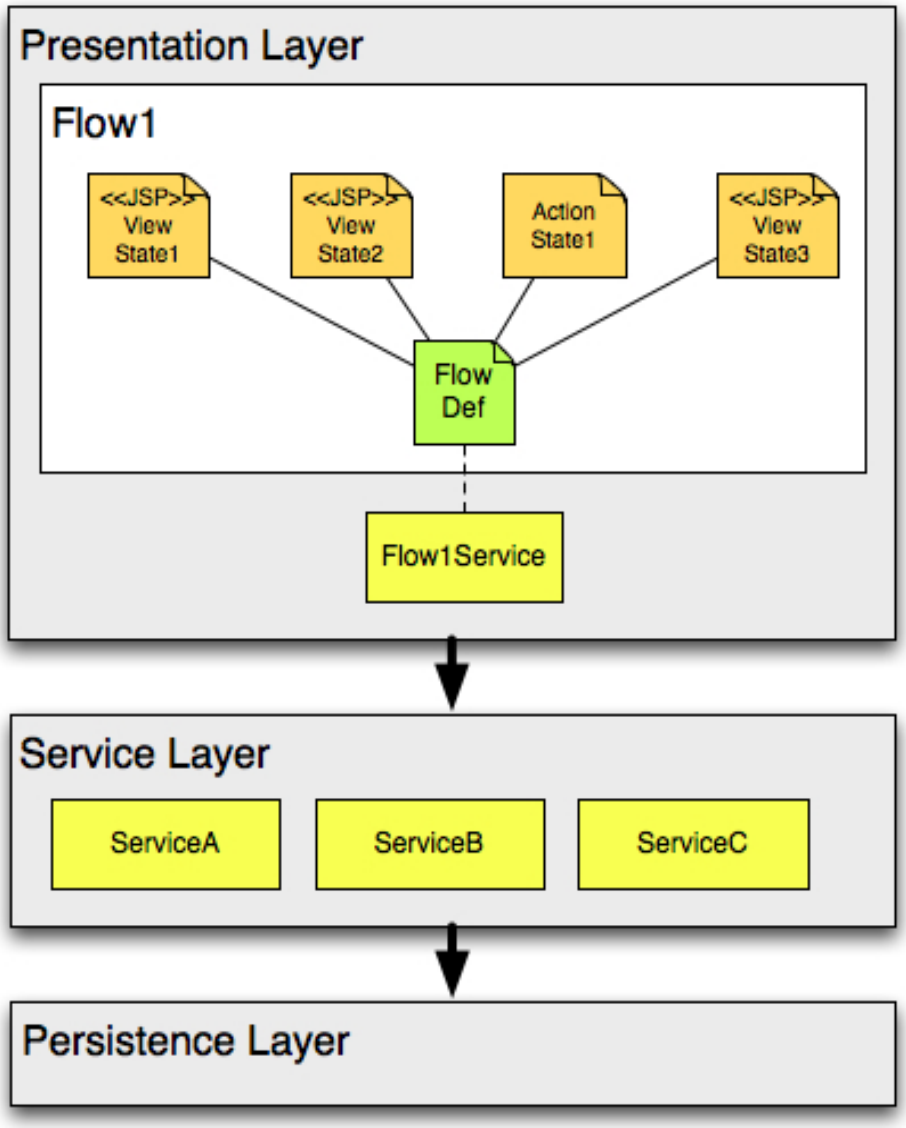


Suggested structure: One presentation-layer service per state in a Spring Web Flow flow.

An appropriate naming scheme should be used to facilitate easy navigation between a state and its corresponding service bean. The Java package hierarchy containing the source code for the service beans can also, for instance, contain a package per flow.

One Presentation Service Per Flow

The second suggestion is more appropriate in flows that have relatively few states or that require a smaller amount of interaction with services. Each Spring Web Flow flow has one corresponding service bean in the presentation layer. All states in the flow invokes methods on one and the same service bean.



Suggested structure: One presentation-layer service per Spring Web Flow flow.

As with the previous suggestion, it may be helpful to consider the naming of the different artifacts as well as the structure of the source-code package hierarchy.

12.2. Development Strategies

This section contains some suggestions regarding development strategies that can be used when developing Spring Web Flow applications. Some suggestions may seem obvious, but bear in mind that this is meant to be an introductory text.

Draw Flow Diagrams

Draw some kind of diagrams, for instance UML activity diagrams, to visualize flows. These diagrams may be thrown away after having finished developing the flow and its associated unit test. This usually helps me to:

- Get a better overview of the flow.
- Discover “loose ends” in the flow.
  For instance, transitions that I have forgot to add.
- Clarifies the structure of the flow.
  This may help me to discover sub-flows.
- Makes writing the unit test of the flow easier.

Structuring the Flow

To me, structuring the flow is an activity closely related to drawing flow diagrams, described in the above section. It helps me to break down the flow into smaller components and see similarities between flows.
Some things to consider in connection to flow structure are:

- Subflows.
  A flow and its subflows share the contents of the conversation scope.
  Calling a subflow from a flow is similar to a synchronous function invocation; the calling flow waits until the subflow has finished and returns before proceeding.
- Flow inheritance.
  A flow may inherit from one or more parent flows.
- State-level inheritance.
  A state in a flow may inherit from a single state of the same type in a (single) parent flow.
- Abstract flow.
  A flow may be abstract, which prevents it from being executed.

Unit Test Flows

Unit testing of Spring Web Flows gives the advantages connected to regular Java unit testing and test-driven development. As we have seen in chapter five and ten, the excellent support in Spring Web Flow 2 makes developing flow unit tests quite easy.
The only thing I miss when testing flows, compared to when testing Java code, is some kind of code coverage tool.

Flow Variable Scoping

As we have seen, there are different scopes in a flow. However, there is nothing that prevents us from using variables with the same names in the different scopes. There is also nothing that prevents us from storing all variables in the flow scope, making them available during the entire lifetime of the flow.
A good practice is, as far as I am concerned, to try to think about when a variable is needed and use the smallest possible scope for the variable. Some advice on scoping are:

Scope	Scope Life-Span
flowScope	Used when the variable must be available during the entire lifetime of the flow.
flashScope	Used when the variable is not needed after the next view render, when the flash scope will be cleared.
requestScope	Used when the variable is to be available during the entire processing of a user request.
viewScope	Used when the variable is to be used only from that a view state is entered to when the view state is exited.
conversationScope	Used when the variable must be available during the entire lifetime of the flow and to all the subflows of the flow.

All Flows Should Have an End State

To give Spring Web Flow a chance to clean up after the completion of a flow and release associated resources, all flows should have an end state.

Statefulness and Thread Safety

Service beans in the presentation layer are regular Spring beans and as such there will be one instance per bean type in the default configuration. These beans are also POJOs and, unless you make all the methods synchronized, multiple threads may be executing simultaneously in a bean. The options in this area are:

- Singleton beans.
  Such beans must be stateless and thread-safe.
  Do not use any instance variables that are not immutable.
  Store state in a flow variables, if required, and enclose it in each service request where it is needed.
- One bean instance per flow.
  Using the prototype scope with the Spring service beans, it is possible to have one instance of a service bean per flow. Such beans may hold state associated with the client.

In both cases, memory usage will rise with an increased number of clients.

Minimize Processing in the Flow Definition

Given the powerful features of Spring Web Flow, it is possible to perform quite some processing in the flow definition (file). However, I would recommend doing the opposite; minimize the amount of processing in the flow definition and push as much as possible of this into the service bean(s) in the processing layer.
This increases readability of the flow definition file and makes refactoring and other maintenance of the code easier, since you will have better tooling for Java code.

One Directory Per Flow

As said both in the Spring Web Flow reference and earlier in this tutorial, it is a good idea to create one directory per flow and store all the resources associated with the flow in that directory.

This concludes this chapter and this tutorial.