Lab 2: JSF Architecture

In this lab, you will write a simple JSF application to learn the basic architecture of the framework.

The application will be for a fictional developer's conference with an online registration system. The application will let the user search for their registration information by supplying the confirmation number.

Objectives:

- To write a complete JSF application
- To understand the fundamentals of the framework

Part 1: Getting Started

Steps:

_1. Your first job is to create a project for the lab. Follow this procedure:

   a. From the Eclipse menu, choose *File - New - Dynamic Web Project* to start the wizard.

   b. On the first wizard page, for the *Project name*, enter **lab02Web**.

      Put a checkmark in the *Add project to an EAR* box, then for the *EAR project name*, enter **lab02EAR**.

      Press Finish. You should see the new projects in the Project Explorer.

   c. In the Project Explorer, right-click on the *lab02Web* project and choose *Properties*.

   d. In the left-hand category list, click on *Project Facets*.

      Then find the entry for *JavaServer Faces* and look to its right side and click the down-arrow icon to set the version to **1.2**.

      Then put a checkmark next to *JavaServer Faces*, but don't close the Properties dialog yet.

   e. At the bottom of the dialog, there should be a link: *Further configuration available*.

      Click the link to bring up the *JSF Capabilities* dialog.

      In the *URL Mapping Patterns*, select the existing entry to highlight it, then press the *Remove* button.

      Press the *Add* button and add a pattern of ***.faces**.

      Press OK to return to the Project Facets dialog.

      Press Apply followed by OK.

_2. Next, create a RegistrationBean class that stores information about a conference registration. Instances of this class don't need to be named managed beans, so you will create the class in a slightly different fashion than before:

   a. In the *lab02Web* project, expand the *Java Resources* folder, then right-click on the *src* folder and choose *New - Class* to start the wizard.

   b. On the first page, for the *Package*, enter **conference**.

      For the *Name*, enter **RegistrationBean**, then press Finish.

      Eclipse creates the class and opens it into the Java editor.

c. Define the following fields:

```
private int confirmationNumber;
private String name;
private int attendees;
```

Use *Source - Generate Getters and Setters* to create get/set methods for all of the fields.

d. Write a read-only property that returns the total amount due:

```
public double getAmountDue()
{
    return attendees * 2000;
}
```

Save and close the file.

_3. In a real-world conference-registration application, we would store the registration list in a database, but to keep things simple, this application will simply store a registration list in memory.

Java Enterprise Edition Web applications can use **application scope** to store items such as the registration list -- application scope is shared amongst the entire Web application.

In this step, you will write a standard JEE Web application **context listener** class that the JEE container calls when the Web application starts. In your listener, you will create a list of Registration beans and store the list at application scope. Follow this procedure:

a. Right-click on the *Java Resources/src* folder and choose *New - Class* to start the wizard.

For the *Package*, enter **listeners**. For the *Name*, enter **MyContextListener**.

For the *Interfaces*, press the Add button and start typing **ServletCont** and then select *ServletContextListener - javax.servlet* and press OK. Press Finish to complete the wizard.

b. Note that the wizard generated a class that implements ServletContextListener and also wrote empty methods for the two interface methods.

Complete the *contextInitialized* method:

```
ServletContext ctx = arg0.getServletContext();

Vector<RegistrationBean> reglist =
    new Vector<RegistrationBean>();

RegistrationBean rb = new RegistrationBean();
rb.setAttendees(3);
rb.setConfirmationNumber(44);
rb.setName("Paul Westerberg");
reglist.add(rb);
rb = new RegistrationBean();
rb.setAttendees(12);
rb.setConfirmationNumber(644);
rb.setName("Patti Smith");
reglist.add(rb);

ctx.setAttribute("reglist", reglist);
```

This code creates a couple of Registration beans, adds them to a list, and then stores the list reference at application scope using the name **reglist** - other parts of your application can access the list using that name.

Note: Use the *Source - Organize Imports* menu to import needed types.

Save and close the file.

_4. Next, you will write a managed bean class that has logic to search the registration list. Follow these steps to create the class:

a. In the *lab02Web* project, expand the *WebContent/WEB-INF* folder, then double-click on *faces-config.xml* to open it into the Faces Configuration editor.

b. At the bottom of the editor pane, click the *ManagedBean* tab.

c. In the list, click on *session* to ensure it's highlighted, then press the Add button to start the wizard.

d. On the first wizard page, select the *Create a new Java class* button, then press Next.

e. On the next page, for the *Package*, enter **service**.

For the *Name*, enter **ProcessRegistrationsBean**, then press Next.

f. On the next page, for the *Name*, enter **processreg**, then press Finish.

g. At the bottom of the editor window, click the *Source* tab and note the XML to define the managed bean.

Save the faces-config.xml file.

_5. Complete the managed bean:

a. In the Project Explorer, expand the *Java Resources/src/service* entry - you should see ProcessRegistrationsBean.java. Double-click on it to bring it into the editor.

b. Define a field in the new class for the confirmation number that the user will enter to search for a registration:

```
private int confirmationNumber;
```

c. Write another field in which you will store a reference to the "found" registration:

```
private RegistrationBean registration;
```

d. Generate get/set methods for the fields and use *Source - Organize Imports* so Eclipse writes the required import statements.

e. Write a searching method:

```
public String findByConfirmationNumber()
{
    String returnStr = "not-found";

    ExternalContext context =
        FacesContext.getCurrentInstance().getExternalContext();

    Map appMap = context.getApplicationMap();
    Vector<RegistrationBean> reglist =
        (Vector<RegistrationBean>)appMap.get("reglist");

    for (RegistrationBean rb : reglist)
    {
        if (rb.getConfirmationNumber() == confirmationNumber)
        {
            registration = rb;
            returnStr = "found";
            break;
        }
    }
    return returnStr;
}
```

This code first retrieves a reference to a Map that represents application scope, then searches the registration list for the specified registration. Note that JSF will call the setConfirmationNumber method with the confirmation number that the user enters into the HTML input form you will write in a moment.

f. Use *Source - Organize Imports*, then save and close the file.

_6. Now it's time to write a JSP that will let the user enter a confirmation number for which to search. Follow this procedure:

a. Right-click on the *WebContent* folder and choose *New - JSP* to create **FindRegistration.jsp** using the *JavaServer Faces (html)* template.

b. At the top of the source, note the *taglib* directives to use the JSF tags:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

Also note that the wizard generate the *f:view* element required by JSF pages.

c. Change the page's title to **Find Registration**.

d. In the *view*, define an <**h:form**> element.

e. In the *form*, define an HTML paragraph, and inside the paragraph, an <**h:inputText**> element into which the user can enter a confirmation number:

```
<p>
   <h:outputLabel for="confirmnum"
      value="Enter confirmation number: " />
   <h:inputText id="confirmnum"
      value="#{processreg.confirmationNumber}"/>
</p>
```

Note that the JSF EL expression references the managed bean you wrote and configured in earlier steps.

f. Define another HTML paragraph, and in it, an <**h:commandButton**> that will act as the "submit" button for the form:

```
<p>
  <h:commandButton value="Find"
    action="#{processreg.findByConfirmationNumber}"/>
</p>
```

Note how the JSF expression references the *findByConfirmationNumber* method you wrote in the ProcessRegistrationList managed bean class.

Save and close the file.

_7. Next, create another JSP named **DisplayRegistration.jsp** that will display the "found" registration.

Within the *f:view* element, use the <**h:outputText**> component to display the four RegistrationBean properties: name, confirmationNumber, attendees and amountDue. Here's an example to help you get started:

```
<p>
   <h:outputText
      value="Name: #{processreg.registration.name}"/>
</p>
```

Note how we can use the JSF expression language to access nested properties.

Note: Be sure to copy the above code to display ALL of the RegistrationBean properties, each within their own paragraph.

_8. Next, create another JSP named **NotFound.jsp** that simply displays an HTML message that the registration wasn't found.

_9. Next, define the *navigation rules* for your application:

a. Open faces-config.xml into the editor.

b. At the bottom of the editor, press the *Navigation Rule* tab.

c. Show the *Palette* view by choosing from the Eclipse menu *Window - Show View - Other - General - Palette.*

d. In the Palette, click the *Page* icon, then click somewhere on the left side of the gridded faces-config.xml window.

Expand the *lab02Web/WebContent* folder, select *FindRegistration.jsp*, then press OK.

Eclipse draws a icon representing the *FindRegistration.jsp* page.

e. Repeat the above step to create an icon for *DisplayRegistration.jsp*, placing the icon to the right and above *FindRegistration.jsp*.

f. Repeat the above step to create an icon for *NotFound.jsp*, placing the icon to the right and below *FindRegistration.jsp*.

g. In the Palette, click on the *Link* icon, then drag a line from *FindRegistration.jsp* to *DisplayRegistration.jsp*, clicking on *DisplayRegistration.jsp* to complete the line. Eclipse draws an arrowed line from *FindRegistration.jsp* to *DisplayRegistration.jsp*.

   In the Palette, click on the *Select* icon, then click on the arrowed line.

   At the bottom of the editor, click the *Properties* tab (choose *Window - Show View - Properties* if you don't see it). In the Properties tab, for the *From Outcome*, enter **found** - Eclipse labels the arrowed line.

   Warning: Be sure to enter "found" into the *From Outcome* field, NOT the *From Action* field.

h. Repeat the above step to create a navigation rule from *FindRegistration.jsp* to *NotFound.jsp*, labeling it **not-found**.

   After completing this step, your screen should look something like Figure 1:

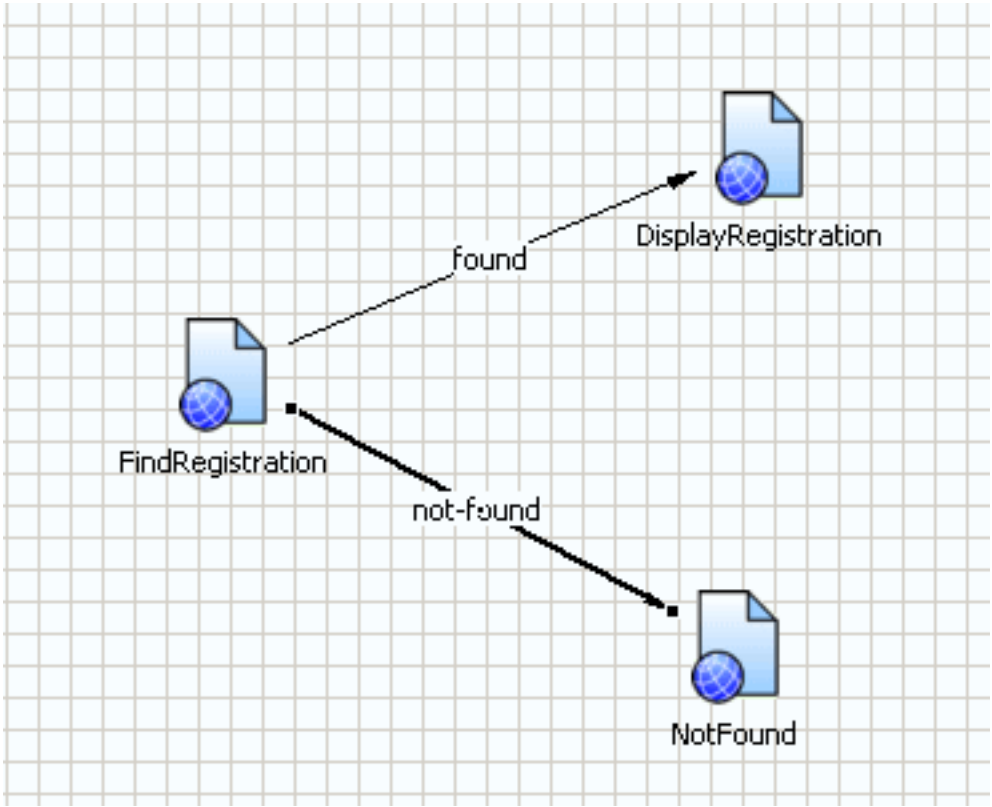

Figure 1: Configuring Navigation Rules

i. In the faces-config.xml editor, click the *Source* tab and examine the navigation rules. They should look like:

```
<navigation-rule>
    <display-name>FindRegistration</display-name>
    <from-view-id>/FindRegistration.jsp</from-view-id>
    <navigation-case>
        <from-outcome>found</from-outcome>
        <to-view-id>/DisplayRegistration.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
<navigation-rule>
    <display-name>FindRegistration</display-name>
    <from-view-id>/FindRegistration.jsp</from-view-id>
    <navigation-case>
        <from-outcome>not-found</from-outcome>
        <to-view-id>/NotFound.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

Save and close the file.

_10. Configure your servlet context listener and welcome file in the *WEB-INF/web.xml* deployment descriptor:

a. Open the deployment descriptor into the editor and then press the *Source* tab at the bottom of the window.

b. Delete entries from the *welcome-file-list* element so that the only entry is *index.jsp*.

c. After the close tag for the *servlet-mapping* element, configure your context listener:

```
<listener>
  <listener-class>
    listeners.MyContextListener
  </listener-class>
</listener>
```

d. Save and close the deployment descriptor editor.

_11. In the Project Explorer, right-click on the *WebContent* folder and choose *New - JSP* to start the wizard to create a "welcome" file for your application:

a. On the first wizard page, for the *File name*, enter **index.jsp**, then press Next.

b. On the next wizard page, uncheck the *Use JSP Template* box, then press Finish.

Eclipse opens the empty JSP into the editor.

c. Add the following, which should be the only thing in the JSP:

```
<jsp:forward page="/FindRegistration.faces"/>
```

Since index.jsp is the "welcome file", when you run the application, this JSP will forward to the actual starting page of the application, which has a URL of *FindRegistration.faces*. JSF requires that all requests from the client use the URL pattern that you configure in web.xml.

_12. Now you can run your program. In the Project Explorer, right-click on the *lab02Web* project and choose

Run As - Run on Server.

Eclipse deploys your application, starts the server and should display your FindRegistration JSP. Enter **44** and press the Submit button with the mouse (pressing Enter doesn't work). You should see the registration that your ContextListener initialized.

Hit the "back" button and try searching for **45** -- you should see your NotFound.jsp.

Hit the "back" button and try searching for **abc** -- what happens? Look in the Eclipse Console pane for an explanation. We will cover validation later.

Part 2: Preventing Direct JSP Access

In this part, you will continue working on the FindRegistration application. JSF requires all access to pages to go through the Faces servlet, using the mapping defined in the web.xml deployment descriptor. In your application, that mapping is URLs that end in ***.faces**.

The problem is that a user may inadvertantly (or on purpose) enter a URL that ends in **.jsp**, which will cause your application to fail if the URL corresponds to an actual JSP. In this part, you will prevent users from directly accessing the JSPs of your pages.

Steps:

_1. Ensure that the application is working, then in the browser's URL field, enter:

```
http://localhost:7001/lab02Web/FindRegistration.jsp
```

This is an attempt to directly access the JSP without going through the Faces servlet. Note that you get an an ugly error page indicating that the FacesContext is not found - that context is supposed to be created by the Faces servlet before it invokes the JSP. Users will likely be dumbfounded by this screen and blame the application rather than thinking they themselves did something wrong.

_2. To prevent the user from directly accessing the JSP, you will write a security constraint:

a. Open web.xml into the editor.

b. After the close tag for the *listener* element, configure a security constraint that prevents client access to the raw JSPs:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Hide JSPs</web-resource-name>
        <url-pattern>/FindRegistration.jsp</url-pattern>
        <url-pattern>/DisplayRegistration.jsp</url-pattern>
        <url-pattern>/NotFound.jsp</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <description>
      Since we define no roles, no direct access
      from the client
  </description>
    </auth-constraint>
</security-constraint>
```

c. Save the file.

_3. In the Servers tab, right-click on the server and choose *Publish* - this ensures that the server is updated with the changes you made to the deployment descriptor.

_4. Go back to the browser window and refresh the page? What happens now?

Note that the user still sees an ugly error page of **Error 403: Forbidden**, but at least this way, the application is protected against direct access and perhaps the user will realize they did something wrong.