4

Developing Web Applications using JavaServer Faces

In the previous two chapters we covered how to develop web applications in Java using Servlets and JSPs. Although a lot of applications have been written using these APIs, most modern Java applications are written using some kind of web application framework. As of Java EE 5, the standard framework for building web applications is Java Server Faces (JSF). In this chapter we will see how using JSF can simplify web application development.

The following topics will be covered in this chapter:

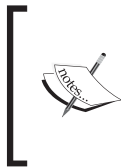
- Creating a JSF project with NetBeans
- Generating a form to capture user data by draging a JSF form from the NetBeans palette into our page
- Laying out JSF tags by taking advantage of the JSF `<h:panelGrid>` tag
- Using static and dynamic navigation to define navigation between pages
- Using the NetBeans **New JSF Managed Bean** wizard to create a JSF managed bean and automatically add it to the application's `<faces-config.xml>` configuration file
- Using the NetBeans Page Flow editor to establish page navigation by graphically connecting pages
- Implementing custom JSF validators
- Displaying tabular data in our pages by dragging-and-dropping the **JSF Data Table** item from the NetBeans palette into our page

Introduction to JavaServer Faces

Before JSF was developed, Java web applications were typically developed using non-standard web application frameworks such as Apache Struts, Tapestry, Spring Web MVC, or many others. These frameworks are built on top of the Servlet and JSP standards, and automate a lot of functionality that needs to be manually coded when using these APIs directly.

Having a wide variety of web application frameworks available (at the time of writing, Wikipedia lists 35 Java web application frameworks, and this list is far from extensive!), often resulted in "analysis paralysis", that is, developers often spend an inordinate amount of time evaluating frameworks for their applications.

The introduction of JSF to the Java EE 5 specification resulted in having a standard web application framework available in any Java EE 5 compliant application server.



> We don't mean to imply that other web application frameworks are obsolete or that they shouldn't be used at all, however, a lot of organizations consider JSF the "safe" choice since it is part of the standard and should be well supported for the foreseeable future. Additionally, NetBeans offers excellent JSF support, making JSF a very attractive choice.

Strictly speaking, JSF is not a web application framework as such, but a component framework. In theory, JSF can be used to write applications that are not web-based, however, in practice JSF is almost always used for this purpose.

In addition to being the standard Java EE 5 component framework, one benefit of JSF is that it was designed with graphical tools in mind, making it easy for tools and IDEs such as NetBeans to take advantage of the JSF component model with drag-and-drop support for components. NetBeans provides a Visual Web JSF Designer that allow us to visually create JSF applications. This tool is discussed in detail in Chapter 6.

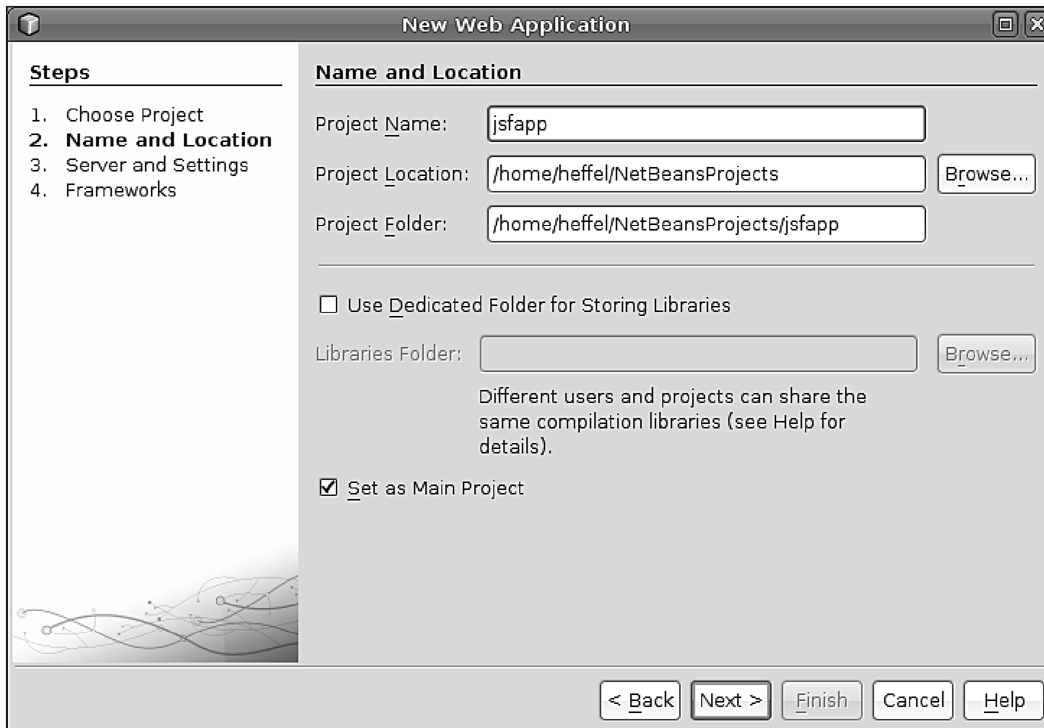
Developing Our first JSF Application

From an application developer's point of view, a JSF application consists of a series of JSP pages containing custom JSF tags, one or more **JSF managed beans**, and a configuration file named `faces-config.xml`. The `faces-config.xml` file declares the managed beans in the application, as well as the navigation rules to follow when navigating from one JSF page to another.

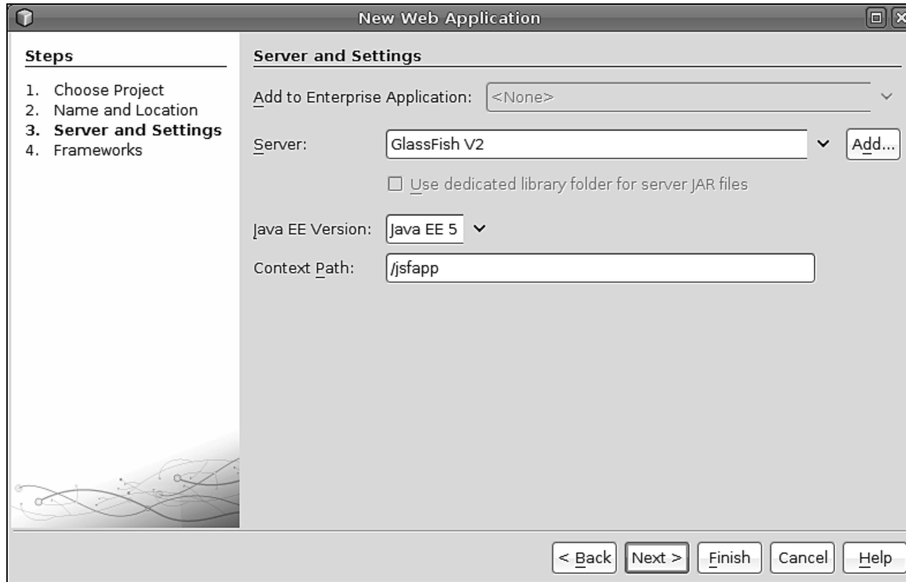
Creating a New JSF Project

To create a new JSF project, we need to go to **File | New Project**, select the **Java Web** project category, and **Web Application** as the project type.

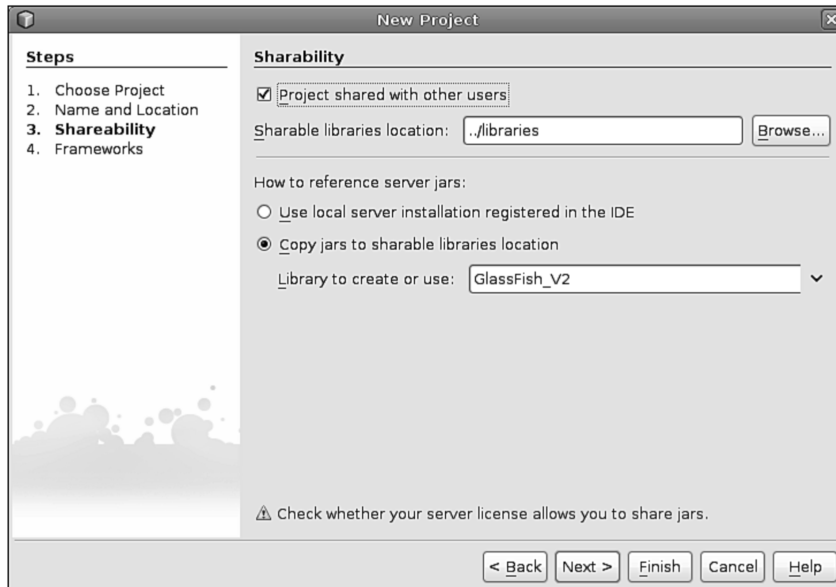
After clicking **Next>**, we need to enter a **Project Name**, and optionally change other information for our project, although NetBeans provides sensible defaults.



On the next page in the wizard, we can select the **Server**, **Java EE Version**, and **Context Path** of our application. In our example we will simply pick the default values.



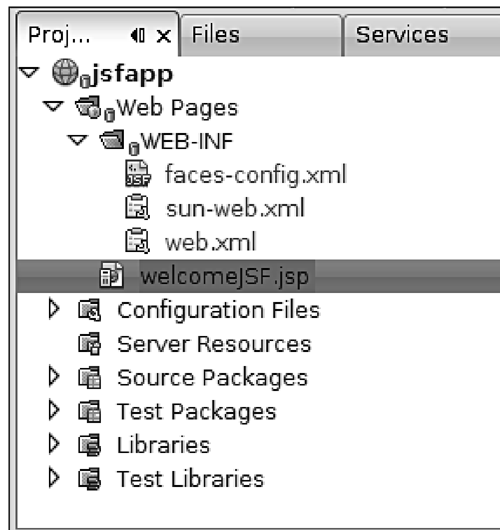
On the next page of the new project wizard, we can select what frameworks our web application will use.



Unsurprisingly, for JSF applications we need to select the JavaServer Faces framework.

The Visual Web JavaServer Faces framework allows us to quickly build web pages by dragging-and-dropping components from the NetBeans palette into our pages. Although it certainly allows us to develop applications a lot quicker than manually coding, it hides a lot of the "ins" and "outs" of JSF. Having a background in standard JSF development will help us understand what the NetBeans Visual Web functionality does behind the scenes. Visual Web JSF is covered in Chapter 6.

When clicking **Finish**, the wizard generates a skeleton JSF project for us, consisting of a single JSP file called `welcomeJSF.jsp`, and a few configuration files: `web.xml`, `faces-config.xml` and, if we are using the default bundled GlassFish server, the GlassFish specific `sun-web.xml` file is generated as well.



web.xml is the standard configuration file needed for all Java web applications. **faces-config.xml** is a JSF-specific configuration file used to declare JSF-managed beans and navigation rules. **sun-web.xml** is a GlassFish-specific configuration file that allows us to override the application's default context root, add security role mappings, and perform several other configuration tasks.

The generated JSP looks like this:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

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

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<%--
    This file is an entry point for JavaServer Faces application.
--%>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
             charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <f:view>
            <h1>
                <h:outputText value="JavaServer Faces"/>
           </h1>
        </f:view>
    </body>
</html>
```

As we can see, a JSF enabled JSP file is a standard JSP file using a couple of JSF-specific tag libraries. The first tag library, declared in our JSP by the following line:

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

is the core JSF tag library, this library includes a number of tags that are independent of the rendering mechanism of the JSF application (recall that JSF can be used for applications other than web applications). By convention, the prefix `f` (for faces) is used for this tag library.

The second tag library in the generated JSP, declared by the following line:

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

is the JSF HTML tag library. This tag library includes a number of tags that are used to implement HTML specific functionality, such as creating HTML forms and input fields. By convention, the prefix `h` (for HTML) is used for this tag library.

The first JSF tag we see in the generated JSP file is the `<f:view>` tag. When writing a Java web application using JSF, all JSF custom tags must be enclosed inside an `<f:view>` tag. In addition to JSF-specific tags, this tag can contain standard HTML tags, as well as tags from other tag libraries, such as the JSTL tags discussed in the previous chapter.

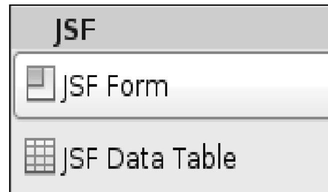
The next JSF-specific tag we see in the above JSP is `<h:outputText>`. This tag simply displays the value of its `value` attribute in the rendered page.

The application generated by the new project wizard is a simple, but complete, JSF web application. We can see it in action by right-clicking on our project in the project window and selecting **Run**. At this point the application server is started (if it wasn't already running), the application is deployed and the default system browser opens, displaying our application's welcome page.

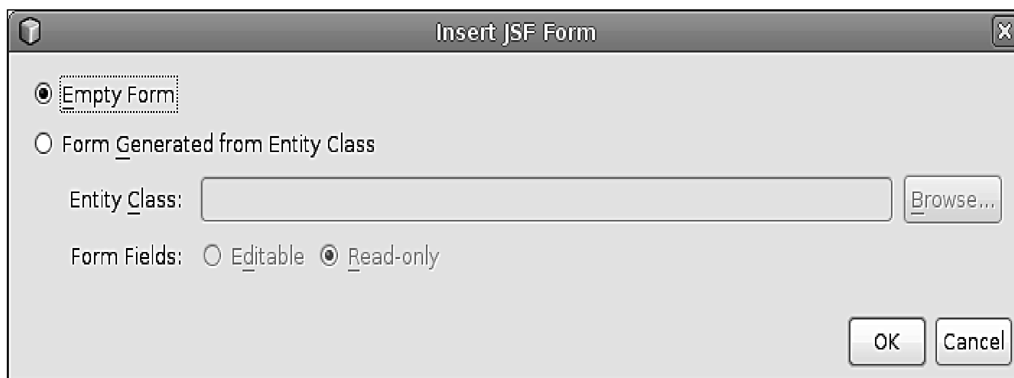
Modifying Our JSP to Capture User Data

The generated application, of course, is nothing but a starting point for us to create a new application. We will now modify the generated `welcomeJSF.jsp` file to collect some data from the user.

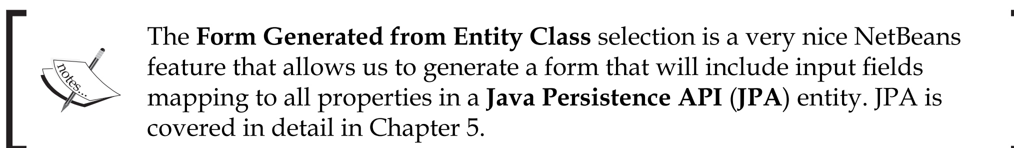
The first thing we need to do is to add an <h:form> tag inside the <f:view> tag. The <h:form> tag is equivalent to the <form> tag in standard HTML pages. We can either type the <h:form> tag directly in the page or drag the **JSF Form** item from the palette into the appropriate place in the page markup.



If we choose the second approach, the following window will pop-up:



Selecting **Empty Form** will generate an empty <h:form> tag which we can use to add our own input fields.



The **Form Generated from Entity Class** selection is a very nice NetBeans feature that allows us to generate a form that will include input fields mapping to all properties in a **Java Persistence API** (**JPA**) entity. JPA is covered in detail in Chapter 5.

After adding the <h:form> tag and a number of additional JSF tags, our page now looks like this:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

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

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type"
        content="text/html; charset=UTF-8">
        <link rel="stylesheet" type="text/css"
        href="../css/style.css">
        <title>JSP Page</title>
    </head>
    <body>
      <f:view>
        <h1><h:outputText value="JavaServer Faces" /></h1>
        <h:form>
          <h:panelGrid columns="3"
            columnClasses="rightalign,leftalign,leftalign">
          <!-- First row begins here -->
          <h:outputLabel value="Salutation: "
             for="salutation"/>
           <h:selectOneMenu id="salutation" label="Salutation"
               value="#{RegistrationBean.salutation}" >
             <f:selectItem itemLabel="" itemValue=""/>
             <f:selectItem itemLabel="Mr." itemValue="MR"/>
             <f:selectItem itemLabel="Mrs." itemValue="MRS"/>
             <f:selectItem itemLabel="Miss" itemValue="MISS"/>
             <f:selectItem itemLabel="Ms" itemValue="MS"/>
             <f:selectItem itemLabel="Dr." itemValue="DR"/>
           </h:selectOneMenu>
           <h:message for="salutation"/>
           <!-- Second row begins here -->
           <h:outputLabel value="First Name:"
              for="firstName"/>
           <h:inputText id="firstName" label="First Name"
               required="true"
              value="#{RegistrationBean.firstName}" />
           <h:message for="firstName" />
           <!-- Third row begins here -->
           <h:outputLabel value="Last Name:" for="lastName"/>
           <h:inputText id="lastName" label="Last Name"
               required="true"
               value="#{RegistrationBean.lastName}" />
           <h:message for="lastName" />
           <!-- Fourth row begins here -->
```

```
                <h:outputLabel for="age" value="Age:"/>
                <h:inputText id="age" label="Age" size="2"
                    value="#{RegistrationBean.age}"/>
                <h:message for="age"/>
                <!-- Fifth row begins here -->
                <h:outputLabel value="Email Address:" for="email"/>
                <h:inputText id="email" label="Email Address"
                    required="true"
                    value="#{RegistrationBean.email}" />
                <h:message for="email" />
                <!-- Sixth row begins here -->
              <h:panelGroup/>
              <h:commandButton id="register" value="Register"
                                          action="submit" />
              </h:panelGrid>
            </h:form>
          </f:view>
        </body>
    </html>
```

The following screenshot illustrates how our page will be rendered at runtime:

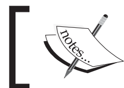
All JSF input fields must be inside a `<h:form>` tag. The `<h:panelGrid>` helps us to easily align JSF tags in our page. It can be thought of as a grid where other JSF tags will be placed. The `columns` attribute of the `<h:panelGrid>` tag indicates how many columns the grid will have, each JSF component inside the `<h:panelGrid>` component will be placed in an individual cell of the grid, when the number of components matching the value of the `columns` attribute (three in our example) has been placed inside `<h:panelGrid>`, a new row is automatically started.

The following table illustrates how tags will be laid out inside a `<h:panelGrid>` tag.

First Tag	Second Tag	Third Tag
Fourth Tag	Fifth Tag	Sixth Tag
Seventh Tag	Eighth Tag	Ninth Tag

Each row in our `<h:panelGrid>` consists of an `<h:outputLabel>` tag, an input field, and an `<h:message>` tag.

The `columnClasses` attribute of `<h:panelGrid>` allow us to assign CSS styles to each column inside the panel grid. Its `value` attribute must consist of a comma separated list of CSS styles (defined in a CSS stylesheet). The first style will be applied to the first column, the second style will be applied to the second column, the third style will be applied to the third column, so on and so forth. If our panel grid had more than three columns, then the fourth column would have been styled using the first style in the `columnClasses` attribute, the fifth column would have been styled using the second style in the `columnClasses` attribute, so on and so forth.



The CSS stylesheet for our example is very simple, therefore it is not shown. However, it is part of the code download for this chapter.

If we wish to style rows in an `<h:panelGrid>`, we can do so with its `rowClasses` attribute, which works the same way that the `columnClasses` works for columns.

`<h:outputLabel>`, generates a label for an input field in the form. The value of its `for` attribute must match the value of the `id` attribute of the corresponding input field.

`<h:message>` generates an error message for an input field. The value of its `for` field must match the value of the `id` attribute for the corresponding input field.

The first row in our grid contains an `<h:selectOneMenu>`. This tag generates an HTML `<select>` tag on the rendered page.

Every JSF tag has an `id` attribute. The value for this attribute must be a string containing a unique identifier for the tag. If we don't specify a value for this attribute, one will be generated automatically. It is a good idea to explicitly state the ID of every component, since this ID is used in runtime error messages (affected components are a lot easier to identify if we explicitly set their IDs).

When using `<h:label>` tags to generate labels for input fields, or when using `<h:message>` tags to generate validation errors, we need to explicitly set the value of the `id` tag, since we need to specify it as the value of the `for` attribute of the corresponding `<h:label>` and `<h:message>` tags.

Every JSF input tag has a `label` attribute. This attribute is used to generate validation error messages on the rendered page. If we don't specify a value for the `label` attribute, then the field will be identified in the error message by it's ID.

Each JSF input field has a `value` attribute. In the case of `<h:selectOneMenu>`, this attribute indicates which of the options in the rendered `<select>` tag will be selected. The value of this attribute must match the value of the `itemValue` attribute of one of the nested `<f:selectItem>` tags. The value of this attribute is usually a **value binding expression**, which means that the value is read at runtime from a JSF-managed bean. In our example, the value binding expression `#{RegistrationBean.salutation}` is used. What will happen is, at runtime JSF will look for a managed bean named `RegistrationBean`, and look for an attribute named `salutation` on this bean, the `getter` method for this attribute will be invoked, and its return value will be used to determine the selected value of the rendered HTML `<select>` tag.

Nested inside the `<h:selectOneMenu>` there are a number of `<f:selectItem>` tags. These tags generate HTML `<option>` tags inside the HTML `<select>` tag generated by `<h:selectOneMenu>`. The value of the `itemLabel` attribute is the value that the user will see, while the value of the `itemValue` attribute will be the value that will be sent to the server when the form is submitted.

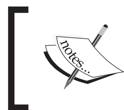
All other rows in our grid contain `<h:inputText>` tags. This tag generates an HTML `input` field of type `text`, which accepts a single line of typed text as input. We explicitly set the `id` attribute of all of our `<h:inputText>` fields; this allows us to refer to them from the corresponding `<h:outputLabel>` and `<h:message>` fields. We also set the `label` attribute for all of our `<h:inputText>` tags; this results in user friendlier error messages.

Some of our `<h:inputText>` fields require a value. These fields have their `required` attribute set to `true`, and each JSF input field has a `required` attribute. If we require the user to enter a value for this attribute, then we need to set this attribute to `true`. This attribute is optional, and if we don't explicitly set a value for it, then it defaults to false.

In the last row of our grid, we added an empty `<h:panelGroup>` tag. The purpose of this tag is to allow adding several tags into a single cell of an `<h:panelGrid>`. Any tags placed inside this tag are placed inside the same cell of the grid where `<h:panelGrid>` is placed. In this particular case, all we want to do is to have an "empty" cell in the grid so that the next tag, `<h:commandButton>`, is aligned with the input fields in the rendered page.

`<h:commandButton>` is used to submit a form to the server. The value of it's `value` attribute is used to generate the text of the rendered button. The value of it's `action` attribute is used to determine what page to display after the button is pressed. This is specified in the navigation rules of the application's `faces-config.xml` file, which will be covered later in the chapter.

In our example, we are using **static navigation**. When using JSF static navigation, the value of the `action` attribute of a command button is hard coded in the JSP markup. An alternate to static navigation is **dynamic navigation**. When using dynamic navigation, the value of the `action` attribute of the command button is a value binding expression resolving to a method returning a `String` in a managed bean. The method may then return different values based on certain conditions. Navigation would proceed to a different page, depending on the value of the method.



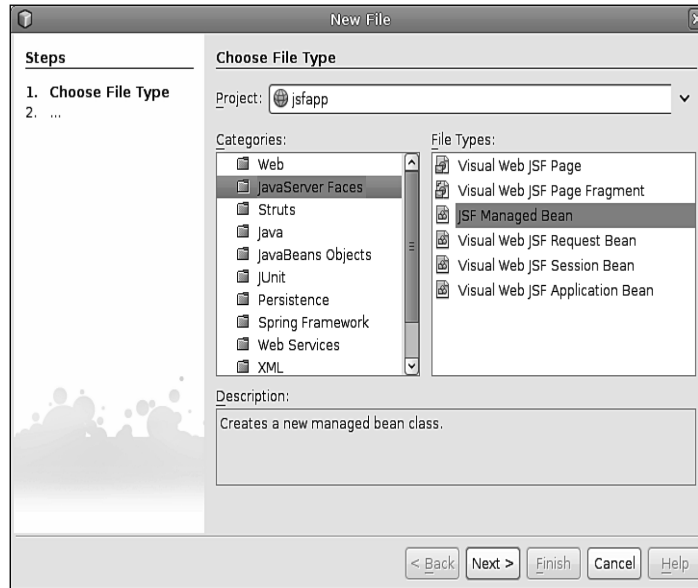
As long as it returns a `String`, the managed bean method executed when using dynamic navigation can contain any logic inside it, and is frequently used to save data in a managed bean into a database.

Both when using static or dynamic navigation, the page to navigate to is defined in the application's `faces-config.xml` configuration file. Later in this chapter, we will explain how we can graphically configure navigation rules using the NetBeans Page Flow editor.

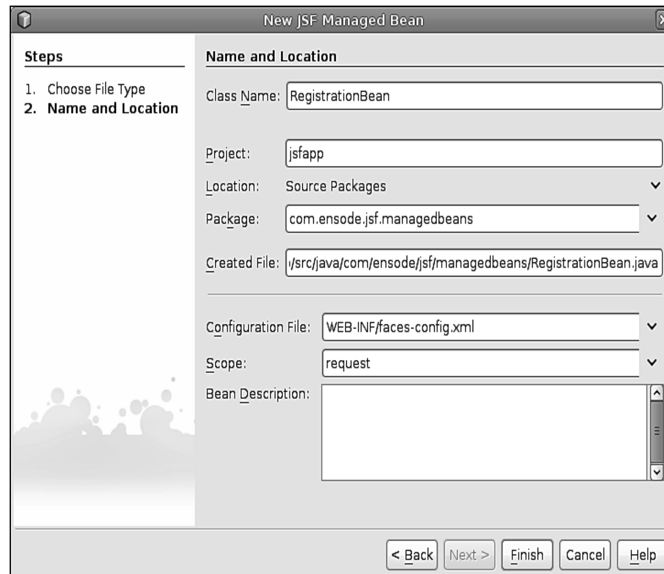
Creating Our Managed Bean

JSF-managed beans are standard JavaBeans that are used to hold user-entered data in JSF applications. JSF-managed beans need to be declared in the application's `faces-config.xml` file. NetBeans can help expedite things by automatically adding our managed beans to `faces-config.xml`.

In order to create a new managed bean, we need to go to **File | New**, select **JavaServer Faces** from the category list, and **JSF Managed Bean** from the file type list.



In the next screen in the wizard, we need to enter a name for our managed bean, as well as a package.



Most default values are sensible and in most cases can be accepted. The only one we should change if necessary is the **Scope** field.

Managed beans can have different scopes. A scope of request means that the bean is only available in a single HTTP request. Managed beans can also have session scope, in which case they are available in a single user's HTTP session. A scope of application means that the bean is accessible to all users in the application, across user sessions. Managed beans can also have a scope of none, which means that the managed bean is not stored at any scope, but is created on demand as needed. We should select the appropriate scope for our managed bean. In our particular example, the default request scope will meet our needs.

After finishing the wizard, two things happen: a boilerplate version of our managed bean is created in the specified package, and our managed bean is added to the application's `faces-config.xml`.

The generated managed bean source simply consists of the class and a public no argument constructor.

```
package com.ensode.jsf.managedbeans;

public class RegistrationBean {

    /** Creates a new instance of RegistrationBean */
    public RegistrationBean() {
    }
}
```

The application's `faces-config.xml` contains our managed bean declaration.

```
<?xml version='1.0' encoding='UTF-8'?>

<faces-config version="1.2"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">
    <managed-bean>
        <managed-bean-name>
         RegistrationBean
       </managed-bean-name>
        <managed-bean-class>
         com.ensode.jsf.managedbeans.RegistrationBean
        </managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
    </managed-bean>
</faces-config>
```

The value of the `<managed-bean-name>` element matches the value we entered in the **Class Name** field in the wizard. Notice that this value is what we used in the value binding expressions in our page to access the managed bean properties. Although the value we gave the managed bean matches it's class name, this is not mandatory.

The value we entered in the wizard's **Class Name** field is also used as the name of the class that was generated by the wizard, as can be seen by the value of the `<managed-bean-class>` element, which is the fully qualified name of our managed bean class. Unsurprisingly, the package structure matches the value we entered in the **Package** field in the wizard. Finally, we see the scope we selected in the wizard as the value of the **<managed-bean-scope>** element.

At this point, we need to modify our managed bean by adding properties that will hold the user-entered values.

Automatic Generation of Getter and Setter Methods

Netbeans can automatically generate getter and setter methods for our properties. We simply need to click the keyboard shortcut for "insert code", which defaults to *Alt+Insert* in Windows and Linux, then select **Getters and Setters**.

```
package com.ensode.jsf.managedbeans;

public class RegistrationBean {

    /** Creates a new instance of RegistrationBean */
    public RegistrationBean() {
    }

    private String salutation;
    private String firstName;
    private String lastName;
    private Integer age;
    private String email;

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
```

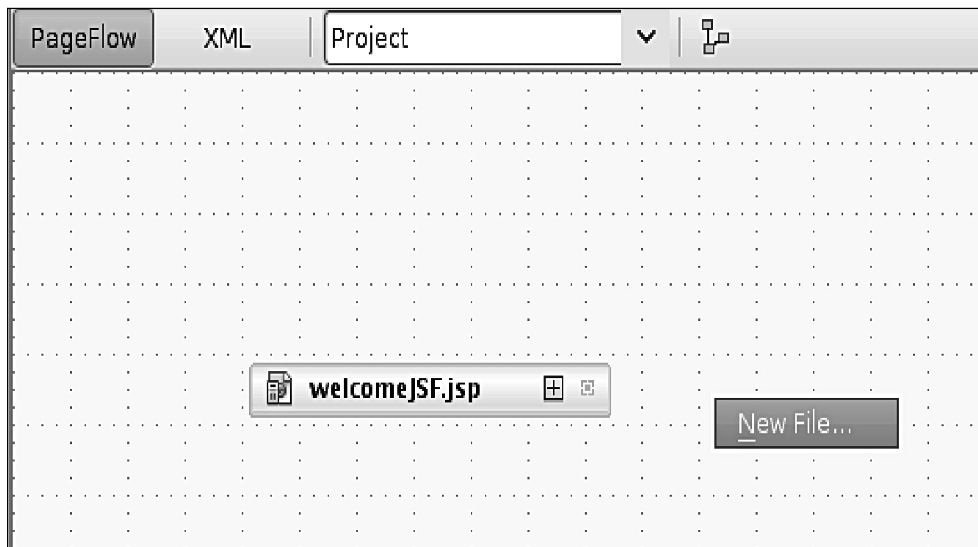
```
    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getSalutation() {
        return salutation;
    }

    public void setSalutation(String salutation) {
        this.salutation = salutation;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}
```

Notice that the names of all of the bean's properties (instance variables) match the names we used in the JSP's value binding expressions. These names must match so that JSF knows how to map the bean's properties to the value binding expressions.

Implementing Navigation

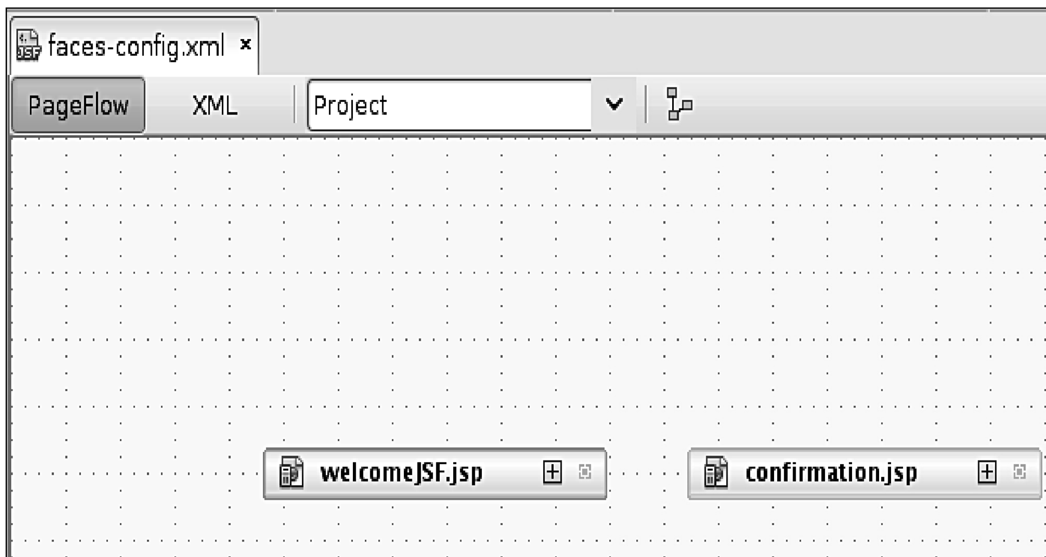
The last thing we need to do before we can test our application is to implement application navigation. For our application we need to create a confirmation page, then add navigation rules to our JSP page so that the application navigates from the input page to the confirmation page when the user submits the form.

NetBeans allows us to save some time by allowing us to graphically add navigation rules via the NetBeans **Page Flow Editor**. To do so, we need to open **faces-config.xml** and click on the **PageFlow** button in the toolbar above the file. In our particular case we haven't yet created the confirmation page we wish to navigate to. This is not a problem, since it can be created "on demand" by NetBeans by right-clicking on the **PageFlow** editor and selecting **New File** from the resulting pop-up menu.

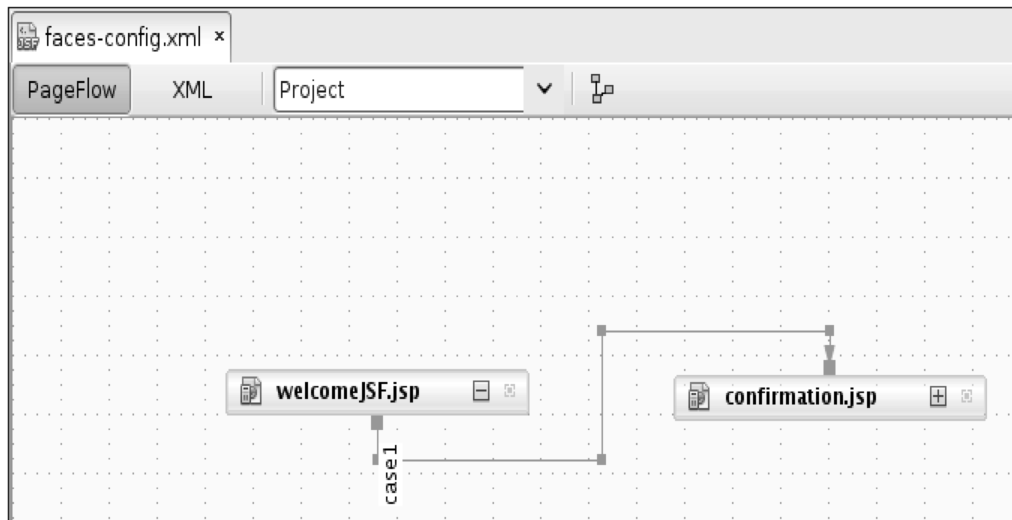


At this point the standard **New JSP File** wizard appears. We enter `confirmation.jsp` as the name of the new JSP. The new page is automatically created and added to the page flow editor.

Refer to Chapter 2 *Developing Web Applications with Servlets and JSPs* for instructions on the **New JSP File** wizard.

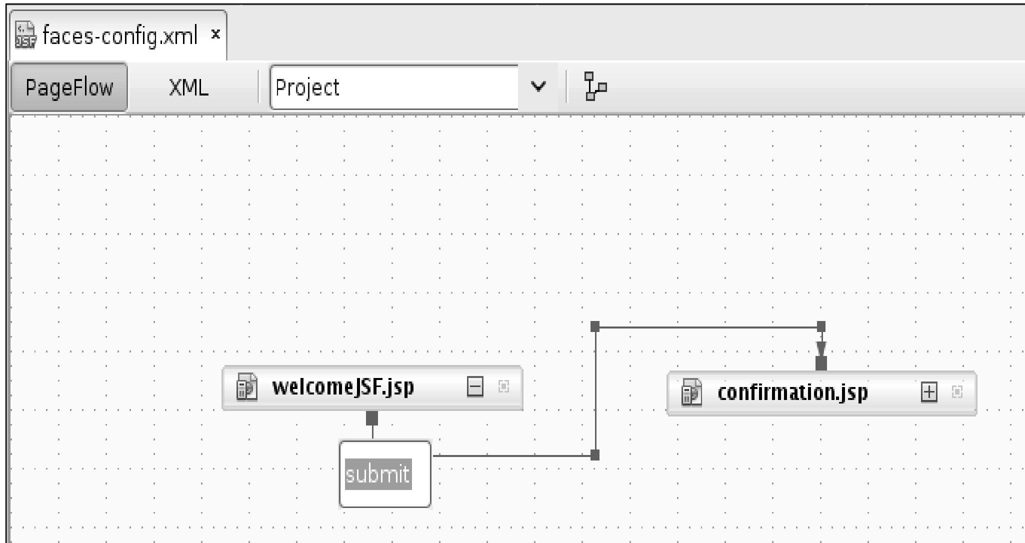


We can graphically connect the two pages by clicking on the connector to the right of **welcomeJSF.jsp** and dragging it to **confirmation.jsp**.

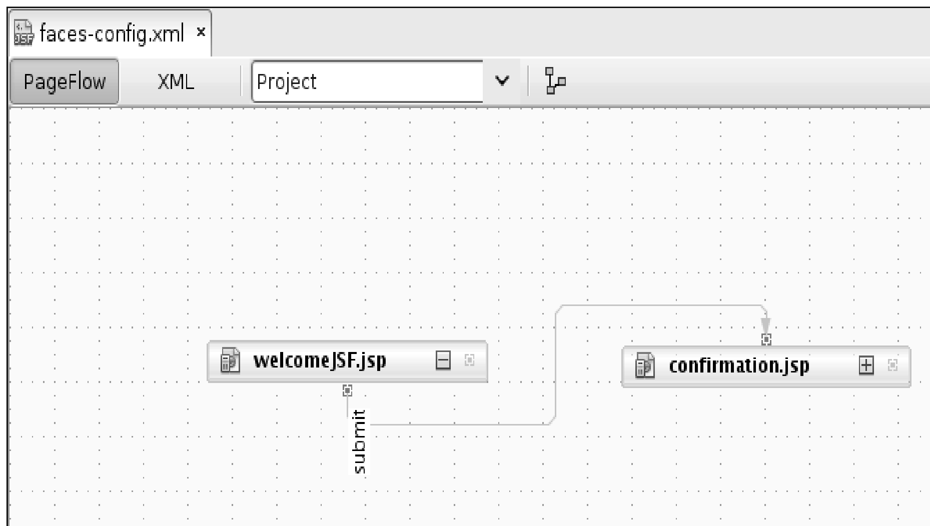


Doing so generates a navigation case from **welcomeJSF.jsp** to **confirmation.jsp**. As we can see, the navigation case is given a default outcome name of **case1**. We need to modify this to be the value of the `action` attribute of the `<h:commandButton>` in `welcomeJSF.jsp`.

To do this, we simply double-click on the text representing the navigation case outcome name, then replace it with the appropriate value.



At this point, the navigation case name is updated with the value we entered.



If we had been using dynamic navigation (and, of course, if there were more than two JSP pages in the application), we would simply drag the connector from **welcomeJSF.jsp** to another page to create a different navigation case based on the value of the managed bean method executed when clicking the page's command button.

The NetBeans **PageFlow** editor updates our application's `faces-config.xml` behind the scenes. It adds a `<navigation-rule>` element to it.

```
<navigation-rule>
    <from-view-id>/welcomeJSF.jsp</from-view-id>
    <navigation-case>
        <from-outcome>submit</from-outcome>
        <to-view-id>/confirmation.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

The `<from-view-id>` element is the name of the JSP originating the navigation. It is the JSP we drag from in the **PageFlow** editor to create the navigation case. The value of the `<to-view-id>` element is the destination page. It is generated from the JSP we drag the navigation case to in the **PageFlow** editor. The value of the `<from-outcome>` element is the name of the navigation case outcome in the **PageFlow** editor.

If we had been using dynamic navigation, we would have separate `<navigation-case>` elements for each possible return value of the managed bean method bound to the page's command button, the body of the `<from-outcome>` element of each navigation case would be one possible return value, and the body of the `<to-view-id>` would be the page we would navigate to for that particular navigation case.

Notice that the value of the `<from-view-id>` element starts with a forward slash (/). A common mistake when setting up JSF navigation is to forget this initial tag. When this happens, JSF will fail to find the destination JSP and will simply redisplay the page that initiated the navigation. Using NetBean's **PageFlow** editor prevents us from making that mistake.

After setting up our navigation case, we now need to modify the generated `confirmation.jsp` so that it displays the values in our managed bean.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<html>
```

```
    <head>
      <meta http-equiv="Content-Type" content="text/html;
          charset=UTF-8">
      <link rel="stylesheet" type="text/css"
          href="../css/style.css">
      <title>Confirmation Page</title>
    </head>
    <body>
      <h2>Confirmation Page</h2>
      <f:view>
        <h:panelGrid columns="2"
            columnClasses="rightalign-bold,normal">
          <!-- First row begins here -->
          <h:outputText value="Salutation: "/>
          <h:outputText
              value="#{RegistrationBean.salutation}" />
          <!-- Second row begins here -->
          <h:outputText value="First Name:"/>
          <h:outputText value="#{RegistrationBean.firstName}" />
          <!-- Third row begins here -->
          <h:outputText value="Last Name:"/>
          <h:outputText value="#{RegistrationBean.lastName}" />
          <!-- Fourth row begins here -->
          <h:outputText value="Age:"/>
          <h:outputText value="#{RegistrationBean.age}"/>
          <!-- Fifth row begins here -->
          <h:outputText value="Email Address:"/>
          <h:outputText value="#{RegistrationBean.email}" />
        </h:panelGrid>
      </f:view>
    </body>
  </html>
```

As we can see, our confirmation page is very simple. It consists of a series of `<h:outputText>` tags containing labels and value binding expressions bound to our managed bean's properties.

Executing Our Application

We are now ready to execute our JSF application. The easiest way to do so is to right-click on **welcomeJSF.jsp** and click on **Run File** in the resulting pop-up menu, or, if our application is set as the main project, we can click directly to the "Run" icon in the tool bar at the top of the IDE.

At this point GlassFish (or whatever application server we are using for our project) will start automatically, if it hadn't been started already, the default browser will open and it will automatically be directed to our page's URL.

After entering some data on the page, it should look something like the following screenshot.

When we click on the **Register** button, our `RegistrationBean` managed bean is populated with the values we entered into the page. Each property in the field will be populated according to the value binding expression in each input field.

At this point JSF navigation "kicks-in", and we are taken to the **Confirmation Page**.

The values displayed in the confirmation page are taken from our managed bean, confirming that the bean's properties were populated correctly.

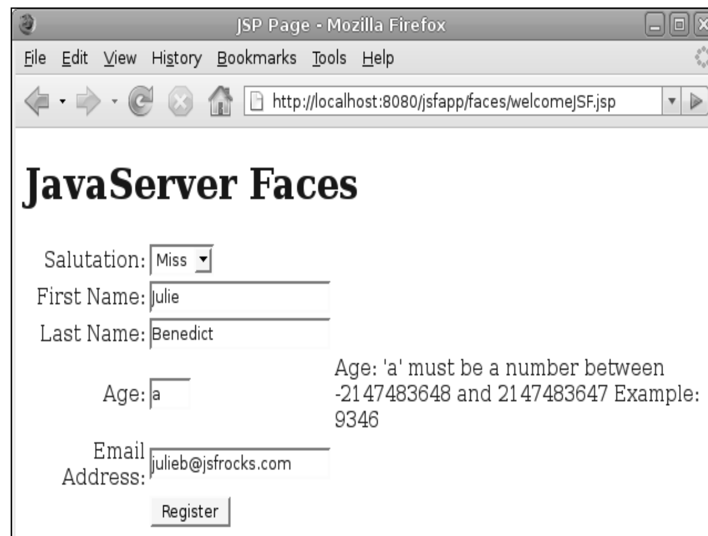
JSF Validation

Earlier in this chapter we discussed how the `required` attribute for JSF input fields allows us to easily make input fields mandatory.

If a user attempts to submit a form with one or more required fields missing, an error message is automatically generated.

The error message is generated by the `<h:message>` tag corresponding to the invalid field. The string `First Name` in the error message corresponds to the value of the `label` attribute for the field. Had we omitted the label attribute, the value of the fields `id` attribute would have been shown instead. As we can see, the `required` attribute makes it very easy to implement mandatory field functionality in our application.

Recall that the `age` field is bound to a property of type `Integer` in our managed bean. If a user enters a value that is not a valid integer into this field, a validation error is automatically generated.



Of course, a negative age wouldn't make much sense, however, our application validates that user input is a valid integer with essentially no effort on our part.

The email address input field of our page is bound to a property of type `String` in our managed bean. As such, there is no built-in validation to make sure that the user enters a valid email address. In cases like this, we need to write our own custom JSF validators.

Custom JSF validators must implement the `javax.faces.validator.Validator` interface. This interface contains a single method named `validate()`. This method takes three parameters: an instance of `javax.faces.context.FacesContext`, an instance of `javax.faces.component.UIComponent` containing the JSF component we are validating, and an instance of `java.lang.Object` containing the user entered value for the component. The following example illustrates a typical custom validator.

```
package com.ensode.jsf.validators;

import java.util.regex.Matcher;
import java.util.regex.Pattern;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.html.HtmlInputText;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;
```

```
public class EmailValidator implements Validator {

    public void validate(FacesContext facesContext,
            UIComponent uIComponent, Object value) throws
            ValidatorException {
        Pattern pattern = Pattern.compile("\\w+@\\w+\\.\\w+");
        Matcher matcher = pattern.matcher(
               (CharSequence) value);
        HtmlInputText htmlInputText = (HtmlInputText) uIComponent;
        String label;

        if (htmlInputText.getLabel() == null ||
                htmlInputText.getLabel().trim().equals("")) {
            label = htmlInputText.getId();
        } else {
            label = htmlInputText.getLabel();
        }

        if (!matcher.matches()) {
            FacesMessage facesMessage =
                    new FacesMessage(label +
                    ": not a valid email address");

            throw new ValidatorException(facesMessage);
        }
    }
}
```

In our example, the `validate()` method does a regular expression match against the value of the JSF component we are validating. If the value matches the expression, validation succeeds, otherwise, validation fails and an instance of `javax.faces.validator.ValidatorException` is thrown.

> The primary purpose of our custom validator is to illustrate how to write custom JSF validations, and not to create a foolproof email address validator. There may be valid email addresses that don't validate using our validator.

The constructor of `ValidatorException` takes an instance of `javax.faces.application.FacesMessage` as a parameter. This object is used to display the error message on the page when validation fails. The message to display is passed as a `String` to the constructor of `FacesMessage`. In our example, if the `label` attribute of the component is not `null` nor empty, we use it as part of the error message, otherwise we use the value of the component's `id` attribute. This behavior follows the pattern established by standard JSF validators.

Before we can use our custom validator in our pages, we need to declare it in the application's `faces-config.xml` configuration file. To do so, we need to add a `<validator>` element just before the closing `</faces-config>` element.

```
<validator>
    <validator-id>emailValidator</validator-id>
    <validator-class>
       com.ensode.jsf.validators.EmailValidator
    </validator-class>
</validator>
```

The body of the `<validator-id>` sub element must contain a unique identifier for our validator. The value of the `<validator-class>` element must contain the fully qualified name of our validator class.

Once we add our validator to the application's `faces-config.xml`, we are ready to use it in our pages.

In our particular case, we need to modify the email field to use our custom validator.

```
<h:inputText id="email" label="Email Address"
        required="true" value="#{RegistrationBean.email}">
    <f:validator validatorId="emailValidator"/>
</h:inputText>
```

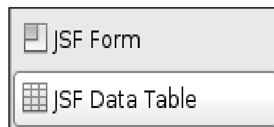
All we need to do is nest an `<f:validator>` tag inside the input field we wish to have validated using our custom validator. The value of the `validatorId` attribute of `<f:validator>` must match the value of the body of the `<validator-id>` element in `faces-config.xml`.

At this point we are ready to test our custom validator.

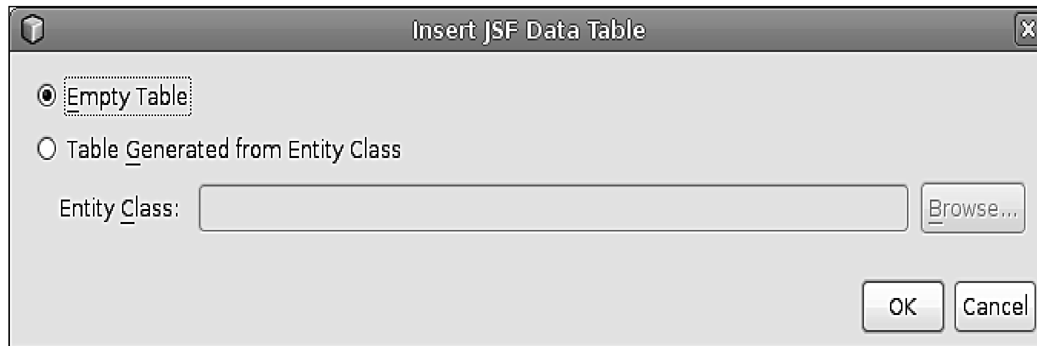
When entering an invalid email address into the email address input field and submitting the form, our custom validator logic was executed and the `String` we passed as a parameter to `FacesMessage` in our `validator()` method is shown as the error text by the `<h:message>` tag for the field.

Displaying Tabular Data

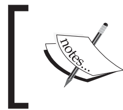
JavaServer Faces includes the `<h:dataTable>` tag that makes it easy to iterate through an array or collection of objects. With NetBeans, a data table tag can be added to a page by simply dragging the **JSF Data Table** item from the NetBeans palette into our page. In order to demonstrate the usage of this tag, let's create a new **Web Application** project, and add a new JSP named `registrationlist.jsp` to it.



After dragging the **JSF Data Table** item into the appropriate location in our `registrationlist.jsp` page, the following window pops up.



We can either select to create an **Empty Table** or a **Table Generated from an Entity Class**.



An Entity Class refers to a Java Persistence API (JPA) entity. We will discuss JPA in detail in Chapter 5 *Interacting With Databases through the Java Persistence API.*

Selecting to create an empty table generates the following markup in our page:

```
<h:form>
  <h:dataTable value="#{arrayOrCollectionOf}"
      var="item">
   </h:dataTable>
</h:form>
```

Notice that NetBeans automatically wraps the generated `<h:dataTable>` tag in an `<h:form>` tag. The `<h:form>` tag is necessary if we plan to have any input fields in our table. Since this is not the case in our example, we can safely delete it.

The value of the `value` attribute of `<h:dataTable>` typically resolves to an array or collection of objects. NetBeans places the placeholder value binding expression `#{arrayOrCollectionOf}` as its value; we must replace this with a value binding expression resolving to one of the appropriate types.

The value of the `var` attribute of `<h:dataTable>` is used to refer to the current element in the table. As we iterate through the elements of the array or collection from the `value` attribute, we can use the value of the `item` attribute to refer to the current element in the array or collection.

We need to add a `<h:column>` tag inside the `<h:dataTable>` tag for each column we wish to add to the table. The following example illustrates typical usage of `<h:dataTable>` and `<h:column>`.

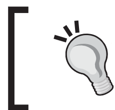
```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
 <html>
   <head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8">
     <title>JSP Page</title>
   </head>
   <body>
  <f:view>
    <h:form>
      <h:dataTable
        value=
         "#{RegistrationListController.registrationBeanList}"
          var="item"  border="1" cellspacing="0"
          cellpadding="5">
        <h:column>
          <f:facet name="header">
            <h:outputText value="Salutation"/>
          </f:facet>
          <h:outputText value="#{item.salutation}"/>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="First Name"/>
          </f:facet>
          <h:outputText value="#{item.firstName}"/>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="Last Name"/>
          </f:facet>
          <h:outputText value="#{item.lastName}"/>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="Age"/>
```

```
            </f:facet>
            <h:outputText value="#{item.age}"/>
          </h:column>
        </h:dataTable>
      </h:form>
    </f:view>
  </body>
   </html>
```

In this example, we will be iterating through a collection of `RegistrationBean` objects. The objects will be stored as a property named `registrationBeanList` of type `java.util.List` in a managed bean called `RegistrationListController`, therefore we set the value of the `value` attribute of `<h:dataTable>` to `#{RegistrationListController.registrationBeanList}`.

NetBeans creates a sensible value for the `var` attribute, therefore we leave it as is.

`<h:dataTable>` contains a few attributes that allow us to control the look of the generated table. These attributes are identical to attributes in a standard HTML table. In our example, we set a border of 1 pixel in the table by setting the value of the `border` attribute to `1`. We set the spacing between table cells to zero by setting the `cellspacing` attribute to `0`. We also set the spacing (padding) inside table cells to 5 pixels by setting the `cellpadding` attribute to `5`.



The complete list of attributes for `<h:dataTable>` can be seen by using code completion (*Ctrl+Space*).

Since our table will have four columns, we need to add four nested `<h:column>` tags into our data table (one for each column).

Notice each `<h:column>` tag has a nested `<f:facet>` tag. JSF tags might define one or more facets. Facets are components that are rendered differently from other components in the parent component. Each facet must have a unique name for each parent component. `<h:column>` defines a facet with a name of `header`, this facet will be rendered as the header of the generated table. To render a facet inside a JSF component, the `<f:facet>` tag is used. In our example we give our facet the name of `header` by assigning this value to its `name` property. At runtime, JSF renders the tag inside `<f:facet>` as the header of the column rendered by the facet's parent `<h:column>` tag. Each `<f:facet>` tag must have a single child tag, which can be any HTML JSF tag.

Adding Multiple Child Components to a Facet
Although the `<f:facet>` tag only accepts a single child component, we can add multiple components to it by nesting them inside an `<f:panelGroup>` tag.

Although not shown in the example, `<h:column>` also defines a facet with a name of `footer` that can be used to render a footer for the column. We simply would add a second facet named `footer` inside our `<h:column>` tag.

Next we add the tags that will be displayed as a single cell for the particular column. We can access the current item in the collection or array. We will be iterating, by using the value of the `var` attribute of `<h:dataTable>` (`item`, in our particular example).

In our example we simply display the values for a single property or each item, however any JSF component can be placed inside `<h:column>`.

Before we can deploy our application and see the above page in action, we need to create the `RegistrationListController` managed bean.

Recall that the easiest way to create JSF managed beans is by going to **File | New**, selecting the **JavaServer Faces** category, and **JSF Managed Bean** as the file type. This procedure is covered in detail earlier in this chapter.

Our managed bean is shown next:

```
package com.ensode.jsf;

import java.util.ArrayList;
import java.util.List;

public class RegistrationListController {

    private List<RegistrationBean> registrationBeanList;

    public RegistrationListController() {
    }

    public String populateList() {
        registrationBeanList = new
        ArrayList<RegistrationBean>();
```

```
            registrationBeanList.add(populateBean(
               "MS", "Carol", "Jones", 35));
            registrationBeanList.add(populateBean(
               "MRS", "Glenda", "Murphy", 39));
            registrationBeanList.add(populateBean(
               "MISS", "Stacy", "Clark", 36));
            registrationBeanList.add(populateBean(
               "MR", "James", "Fox", 40));
            registrationBeanList.add(populateBean(
               "DR", "Henry", "Bennett", 53));

            return "success";
        }

        public List<RegistrationBean> getRegistrationBeanList() {
            return registrationBeanList;
        }

        public void setRegistrationBeanList(
           List<RegistrationBean> registrationBeanList) {
            this.registrationBeanList = registrationBeanList;
        }

        private RegistrationBean populateBean(String salutation,
                String firstName, String lastName, Integer age) {
            RegistrationBean registrationBean;

            registrationBean = new RegistrationBean();
            registrationBean.setSalutation(salutation);
            registrationBean.setFirstName(firstName);
            registrationBean.setLastName(lastName);
            registrationBean.setAge(age);

            return registrationBean;
        }
    }
```

Notice that the bean has a property named `registrationBeanList` of type `java.util.List`. This is the property we used as the value of the `value` property in the `<h:dataTable>` tag in the page above.

The bean's `populateList()` method will be called from another JSP via dynamic navigation. This method populates the `registrationBeanList` property in the bean.

A real application would more than likely retrieve this information from a relational database. To keep our example simple we are simply populating the bean from new instances of `RegistrationBean` we create on the fly.

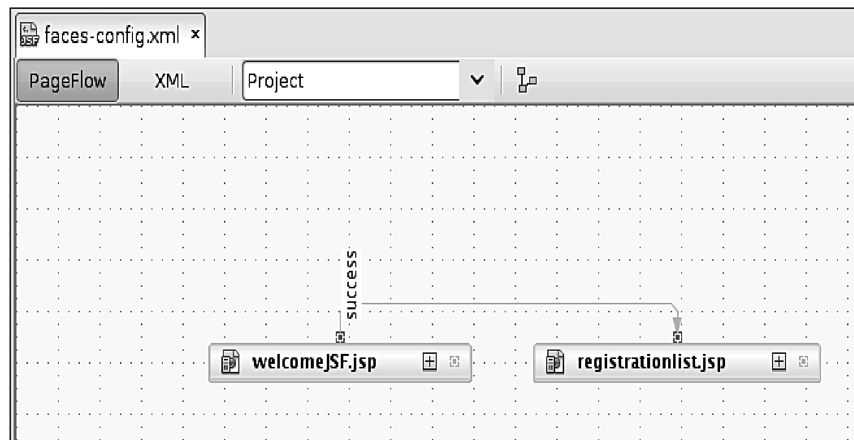
Now we need to modify the JSP that will be invoked initially. When creating the project, NetBeans automatically sets it up so that a JSP called `welcomeJSF.jsp` will be invoked when we point the browser to our application's URL. We need to modify this JSP so that it will invoke the `populateList()` method of our `RegistrationBeanList` managed bean when navigating to the page we wrote earlier.

```
<f:view>
  <h:form>
    <h:commandLink value="Populate List"
      action="#{RegistrationListController.populateList}" />
  </h:form>
</f:view>
```

For brevity, we are only showing the relevant parts of the JSP. Our JSP will have an `<h:commandLink>` tag used for navigation. `<h:commandLink>` is functionally equivalent to `<h:commandButton>`, the main difference is that it is rendered as a link as opposed to a button. The value of the `value` attribute of `<h:commandLink>` is used to render the link text; its `action` attribute is used for navigation. In this case we are using dynamic navigation. When using dynamic navigation, a value binding expression is used as the value of the `action` attribute. This value binding expression must resolve to a method that returns a `String`. The method must take no arguments. When using dynamic navigation, the method that the value binding expression resolves to may return different strings depending on its logic. We can have a page navigate to different pages depending on the value returned by this method. To do this we would have to add a `<navigation-case>` element for each possible value that the method may return.

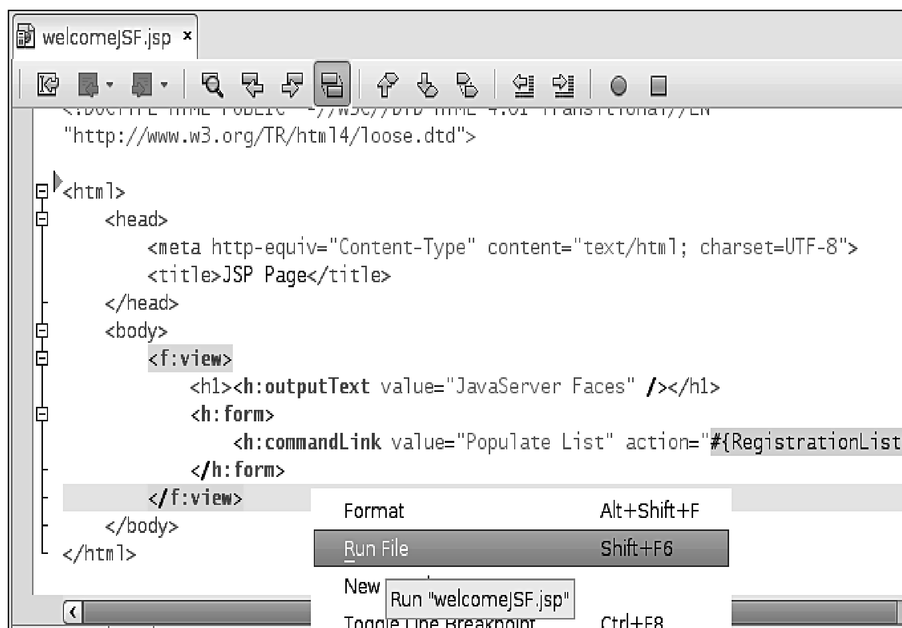
In our example, the `populateList()` method of the `RegistrationListController` managed bean is invoked when a user clicks on the link. This method populates the list that we will iterate through and returns the value of `success`.

Before we deploy our application, we need to define the navigation between our two pages. Normally this is done by manually editing **faces-config.xml**. However, when using NetBeans, it can be done graphically in the NetBeans **PageFlow** editor as explained earlier in this chapter.



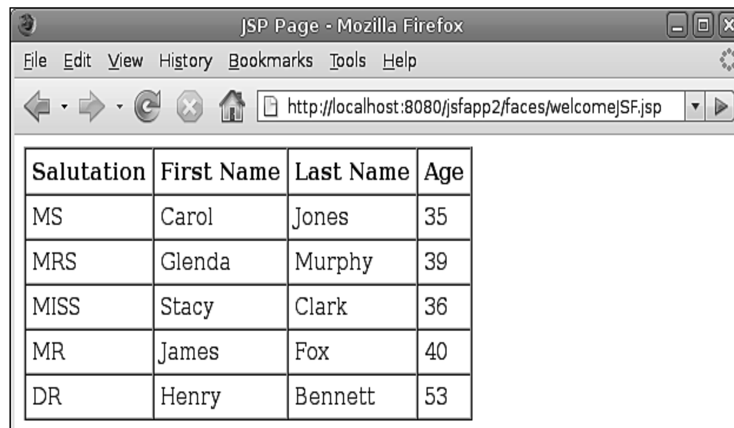
The above screenshot shows the **PageFlow** editor after connecting the initial page containing the `<h:commandLink>` that initiates navigation to the page that iterates through the list of `RegistrationBean` instances, and after changing the default navigation case to `success`. Notice that the text in the navigation case matches the return value of the `populateList()` method in the `RegistrationListController` method. This is how the navigation case is linked to the method's return value.

At this point we are ready to test our application. We can execute the initial `welcomeJSF.jsp` page by right-clicking on it and selecting **Run File**.



At this point, the application server is started (if it hadn't been started previously), the application is deployed, and a browser window opens displaying the page.

Here we can see the link that was generated by the `<h:commandLink>` tag in our JSP. Clicking on that link results in executing the `populateList()` method in the `RegistrationListController` managed bean and navigating to the JSP containing the `<h:dataTable>` tag.



Here we can see the table generated by `<h:dataTable>`, the headers (**Salutation**, **First Name**, **Last Name**, and **Age**) are generated by the `<f:facet>` tags inside each `<h:column>`. While iterating through the collection of `RegistrationBean` objects in the `registrationBeanList` property of the `RegistrationListController` managed bean, each cell in each row displays the property corresponding to the `<c:outputText>` tag inside each `<c:column>` tag in the table.

Summary

In this chapter we saw how NetBeans can help us easily create new JSF projects by automatically adding all required libraries and configuration files.

We also saw how we can create JSF forms for data input and data tables for displaying tabular data by simply dragging and dropping icons from the NetBeans palette into our page.

Additionally, we saw how NetBeans can simplify and significantly speed up development of JSF applications by automatically adding managed bean definitions to the application's `<faces-config.xml>` configuration file, and by allowing us to graphically define navigation rules by taking advantage of the NetBeans **PageFlow** editor.