

FREE EXCERPT

JAVA SERVER FACES IN ACTION

Kito D. Mann

Foreword by Ed Burns

 MANNING



To order this book, go to www.manning.com/mann
or visit your favorite bookstore.



JavaServer Faces in Action
by Kito Mann
Chapter 8 excerpt

Copyright 2005 Manning Publications

brief contents

PART 1 EXPLORING JAVASERVER FACES 1

- 1 ■ Introducing JavaServer Faces 3
- 2 ■ JSF fundamentals 38
- 3 ■ Warming up: getting around JSF 88
- 4 ■ Getting started with the standard components 137
- 5 ■ Using the input and data table components 185
- 6 ■ Internationalization, validators, and converters 234

PART 2 BUILDING USER INTERFACES 275

- 7 ■ Introducing ProjectTrack 277
- 8 ■ Developing a user interface without Java code:
the Login page 287
- 9 ■ Developing a user interface without Java code:
the other pages 316
- 10 ■ Integrating application functionality 354

PART 3 DEVELOPING APPLICATION LOGIC 407

- 11 ■ The JSF environment 409
- 12 ■ Building an application: design issues and foundation classes 456
- 13 ■ Building an application: backing beans, security, and internationalization 499
- 14 ■ Integrating JSF with Struts and existing applications 568

PART 4 WRITING CUSTOM COMPONENTS, RENDERERS, VALIDATORS, AND CONVERTERS 603

- 15 ■ The JSF environment: a component developer's perspective 605

PART 5 WRITING CUSTOM COMPONENTS, RENDERERS, VALIDATORS, AND CONVERTERS: EXAMPLES 703

- 16 ■ UIInputDate: a simple input component 705
- 17 ■ RolloverButton renderer: a renderer with JavaScript support 727
- 18 ■ UIHeadlineViewer: a composite, data-aware component 756
- 19 ■ UINavigator: a model-driven toolbar component 794
- 20 ■ Validator and converter examples 839

ONLINE EXTENSION

The five chapters in part 5 (plus four additional appendixes) are not included in the print edition. They are available for download in PDF format from the book's web page to owners of this book. For free access to the online extension please go to www.manning.com/mann.

Developing a user interface without Java code: the Login page

This chapter covers

- Basic application configuration
- Building a login page with JavaScript, CSS, validators, and custom messages
- Using `HtmlPanelGrid` for layout

Now that we've introduced the case study, ProjectTrack, and its system requirements, let's start building the application. In this chapter and the next, we'll develop ProjectTrack's user interface (UI) with JavaServer Faces. Our goal is to have an interactive interface *without* any application logic. This concept should be familiar to rapid application development (RAD) developers—with tools like AWT/Swing IDEs, Delphi, and Visual Studio .NET, it's quite easy to lay out the entire UI with basic navigation and validation but no real code. Integrating the real meat of the application later is fairly straightforward. The closest thing to this for web developers is typically a set of static HTML pages that eventually get thrown away or converted into application templates and then maintained separately.

Like UIs developed with the tools we just mentioned, the one we develop in this chapter will provide basic navigation and will be the basis for the final version. The main difference is that instead of integrating the UI with application logic and model objects, we'll start with static text and hardcode the navigation rules. Even though JSF tools allow you to work visually and generate a lot of the code for you, we'll show all of the examples in raw JSP so that you can get a feel for what vanilla Faces development is like.

If you're not particularly interested in the JSP aspects of these chapters, bear in mind that the behavior of the standard components is the same even if you use another display technology; the mechanism for declaring the view would be the primary difference.

Developing the UI separately gives us two main benefits:

- The front-end developer can initially work independently from the application developer. We're not suggesting that the two work in a vacuum; however, communication can be limited to discussions of interface points. (See chapter 10 for a discussion of the integration process.)
- The working UI can serve as a prototype that can be quickly modified based on user input; this saves time that would otherwise be wasted creating separate sets of HTML.

We'll start with the basic configuration for the web application. Next, we'll assemble the Login page step by step, examining the fundamental aspects of constructing a JSF page. Then, in the next chapter, we'll develop the remaining pages of ProjectTrack.

8.1 Getting started

Even though we're not writing any Java code in this chapter, we need to create a JSF application so the framework can do its magic. All JSF applications are standard Java web applications. All that's required is a properly configured deployment descriptor and an installed JSF implementation. (This will either be included with your application server or installed separately, as is the case with the reference implementation [Sun, JSF RI].) However, most applications also require the JSF configuration file, `faces-config.xml`, and this one is no different.

Web applications have a specific directory structure. (Tools will often create the basic structure for you.) ProjectTrack's initial directory structure, complete with the basic files, is shown in figure 8.1. You may have noticed that this is a specific instance of the basic directory structure shown in chapter 3 (figure 3.1, page 91).

All JSPs will be placed in the root directory. The deployment descriptor (`web.xml`) and `faces-config.xml` are both placed in the `WEB-INF` directory (no file in this directory is visible to users). The code is placed in the `WEB-INF/lib` directory. Usually, the JSF libraries will be in this directory as well. The structure also includes a directory for images; we listed all of the image filenames based on the assumption that a wildly productive graphic designer has already created them all. The `ptrackResources.properties` will be used to customize application messages and perform localization.

Now that we know where the files will go, let's move on to configuration.

8.1.1 Setting up `web.xml`

Every Java web application has a deployment descriptor called `web.xml`. You can do a lot with this file—specify event handlers for the servlet lifecycle (which is

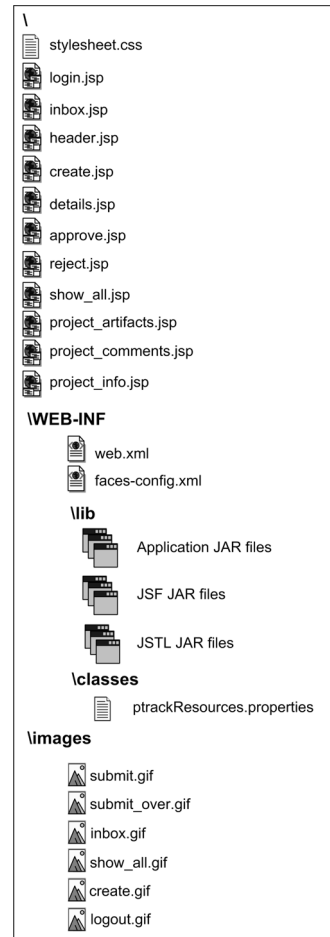


Figure 8.1 ProjectTrack is a web application, whose directory structure includes JSPs, libraries, images, and configuration files.

separate from the JSF Request Processing Lifecycle), configure security, integrate with EJBs, and all sorts of other fun things that we don't discuss in this book. All that's necessary for a simple JSF application like ProjectTrack is a declaration of the `FacesServlet` and the appropriate URL mapping, as shown in listing 8.1. (We added a default page for good measure.)

Listing 8.1 Deployment descriptor (`web.xml`) for UI development

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>ProjectTrack</display-name>
  <description>JavaServer Faces in Action sample application.
</description>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>faces/login.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

The `<welcome-file-list>` element is used to specify the default pages for the web application. For ProjectTrack, the default page is `login.jsp`. Note the “`faces/`” prefix—this ensures that the `FacesServlet` handles the request for the page.

You may have noticed that this deployment descriptor looks a lot like the one for the Hello, world! example from the first chapter. Other than the welcome file, only the name and description are different; the basic requirements are always the same.

8.1.2 Setting up `faces-config.xml`

In chapter 3, we explained how to configure navigation rules. Listing 8.2 defines a single rule for navigating from `login.jsp` (which is the page we'll develop in this chapter).

Listing 8.2 faces-config.xml: ProjectTrack's configuration file with a single navigation rule

```
<?xml version="1.0"?>

<!DOCTYPE faces-config PUBLIC
    "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
  <navigation-rule>
    <from-tree-id>/login.jsp</from-tree-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-tree-id>/inbox.jsp</to-tree-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>failure</from-outcome>
      <to-view-id>/login.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

This navigation rule simply says “For login.jsp, if the outcome is 'success', display inbox.jsp; if the outcome is 'failure', redisplay login.jsp.” When we build inbox.jsp page in this chapter, we’ll hardcode the outcome "success" for our Submit button so that navigation will work without any application code. We’ll continue the process of adding navigation rules and hardcoding outcomes as we walk through additional pages in the next chapter. Then, in chapter 10, we’ll update our component declarations to reference action methods instead.

There’s a lot more to JSF configuration than just navigation rules. As we build ProjectTrack, we’ll define managed beans and configure custom validators, converters, and components. If you want to know more about setting up a JSF environment or configuration in general, see chapter 3.

8.2 Creating the Login page

Let’s get started with the Login page: login.jsp. We’ll build this page step by step, examining various aspects of JSF as we go. We’ll then apply knowledge gained from this process as we build the remaining pages.

The page simply displays a welcome graphic and the name of the application, accepts the username and password for input, and has a single Submit button (because we display only two fields, we have no real need for a Clear button). The

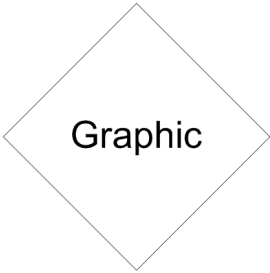
	ProjectTrack	
	Enter name:	<input type="text"/>
	Password:	<input type="password"/>
		<input type="submit" value="Submit"/>

Figure 8.2 Mock-up of the Login page.

password field shouldn't display the text to the user—it should render as a standard HTML password input control so that the browser will show users asterisks instead of the actual text the user types. Figure 8.2 is a rough mock-up of the page.

Structurally, you can think of this design as having a table with one row and two columns: one for the graphic and one for a smaller, embedded table that contains all of the other controls. The embedded table has four rows. The first row has one column and simply contains the title. The second and third rows have two columns that contain a text label and an input field. The final row has two columns: the first is empty, and the second contains a single Submit button.

Now that we have our application configured, and a mock-up of the page, let's get started.

8.2.1 Starting with *HtmlGraphicImage* and *HtmlOutputText* components

We'll begin by creating a basic page that just has the graphic and the text "ProjectTrack" in a large font, as shown in figure 8.3. The JSP is shown in listing 8.3.

Listing 8.3 The JSP source for the Login page with only a graphic and the text "Project Track"

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

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

① Core JSF tag library

② JSF HTML tag library

```
<f:view> ← 3 View tag that
<html>      encloses all
            components

<head>
  <title>
    <h:outputText value="ProjectTrack" /> ← 4 HtmlOutputText
    </title>                                component
  </head>

<body>

<table> ← 5 Table for
<tr>      layout
<td>
  <h:graphicImage url="/images/logo.gif"
    alt="Welcome to ProjectTrack"
    title="Welcome to ProjectTrack"
    width="149" height="160" /> ← 6 HtmlGraphicImage
                                component

</td>
<td>
  <font face="Arial, sans-serif"
    size="6" > ← 7 Format for
                the heading

    <h:outputText value="ProjectTrack" /> ← 8 Another
    </font>                                HtmlOutputText
    </td>                                    component
  </tr>
</table>

</body>
</html>
</f:view>
```



Figure 8.3
Our Login page with only
HtmlGraphicImage
and HtmlOutputText
components.

- ❶ All JSP-based JSF pages must import the core JSF tag library, which includes tags without rendering functionality, like `<f:view>`, and the various validator and converter tags. If we were using a render kit other than the standard one included with JSF, we would still include this tag library.
- ❷ We'll be using the standard HTML components for this application, which is referenced by this tag library.
- ❸ All Faces tags must be enclosed in a `<f:view>` tag.
- ❹ Here we create an `HtmlOutputText` component with the text "ProjectTrack".
- ❺ For now, we'll use standard HTML tables for all of our layout.
- ❻ Here we create an `HtmlGraphicImage` component that references the image located at `/images/logo.gif`. Because the image URL has a leading slash, it will be rewritten to be relative to the web application's root ("`/projects`" in this case). The additional properties—`alt`, `title`, `width`, and `height`—are passed through to the HTML output (which is the `` tag in this case). Most of the tags in the standard HTML tag library behave in this way.
- ❼ For now, we'll use a standard HTML `` tag to format our heading.
- ❽ This is another `HtmlOutputText` component with the text "ProjectTrack". Because it's enclosed in the `` tag described in ❹, it will appear in a larger font.

The corresponding HTML output is shown in listing 8.4.

Listing 8.4 Output of the Login page with only `HtmlGraphicImage` and `HtmlOutputText` components

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>
      ProjectTrack ← HtmlOutputText
    </title>           component
  </head>
  <body>

    <table>
      <tr>
        <td>
           | HtmlGraphicImage
                                                component
        </td>
        <td>
          <font face="Arial, sans-serif" size="6">
            ProjectTrack ← HtmlOutputText
          </font>         component
        </td>
      </tr>
    </table>
  </body>
</html>

```

```
<tr>
</table>

</body>
</html>
```

This was a good start. Let's move on to the meat of the page.

8.2.2 Adding a form

Now we'll add the basic form elements so that we can capture the login name and password. These include an `HtmlInputText` component for the username, an `HtmlInputSecret` component for the password, and an `HtmlCommandButton` component for the Submit button. All of these elements will be placed within an `HtmlForm` so that JSF knows it will be sending the data back to the server. This page uses nested tables to achieve the layout shown in figure 8.2. Figure 8.4 shows the resulting page, displayed in a browser.

The JSP source is shown in listing 8.5.



Figure 8.4 The Login page with input components and a table layout.

Listing 8.5 The JSP source for the Login page with input components and a table layout

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

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

<f:view>
<html>

<head>
  <title>
    <h:outputText value="ProjectTrack"/>
  </title>
</head>

<body>

<h:form> ← ❶ HtmlForm containing
input controls
<table cellpadding="0" cellspacing="0"> ← ❷ Table-based
layout
  <tr>
    <td>
      <h:graphicImage url="/images/logo.gif"
        alt="Welcome to ProjectTrack"
        title="Welcome to ProjectTrack" width="149"
        height="160"/>
    </td>
    <td>
      <table cellpadding="5" cellspacing="3">
        <tr>
          <td colspan="2">
            <font face="Arial, sans-serif" size="6">
              <h:outputText value="ProjectTrack"/>
            </font>
          </td>
        </tr>
        <tr>
          <td>
            <h:outputLabel for="userNameInput">
              <h:outputText
                value="Enter your user name:"/>
            </h:outputLabel>
          </td>
          <td>
            <h:inputText id="userNameInput" size="20"
              maxLength="30"/>
          </td>
        </tr>
      </table>
    </td>
  </tr>
</table>
  <tr>
    <td>
      <h:outputText
        value="Enter your user name:"/>
    </td>
    <td>
      <h:inputText id="userNameInput" size="20"
        maxLength="30"/>
    </td>
  </tr>
</tr>
</table>

```

❸ **HtmlOutputText rendered as a label**

❹ **HtmlInputText with an id**

```
<h:outputLabel for="passwordInput">
  <h:outputText value="Password:"/>
</h:outputLabel>
</td>
<td>
  <h:inputSecret id="passwordInput"
                 size="20"
                 maxlength="20" />
</tr>
</tr>
<tr>
  <td/>
  <td>
    <h:commandButton action="success"
                     title="submit"
                     value="Submit" />
  </tr>
</table>
</td>
</tr>
</table>

</h:form>

</body>
</html>
</f:view>
```

5 **HtmlInputSecret**

6 **HtmlCommandButton**

- 1 We start by declaring an `HtmlForm` component. This will serve as a container for all of the `Input` and `Command` components on the page.
- 2 The table layout matches the layout of the mock-up in figure 8.2, a large table with a single row and two columns. The graphic is in the first column, and a nested table containing the other controls is in the second column.
- 3 Here we use the `<h:outputLabel>` tag to render all embedded components as a label for the control with the identifier “`userNameInput`”. Any component referenced by the `for` property of this tag must have a human-specified identifier. The embedded component is an `HtmlOutputText` instance that has the text “Enter your user name:”. So this code outputs an HTML label for a component called “`userNameInput`” with the value “Enter your user name:”. The same method is used for labeling the password field in the page.
- 4 This code creates a simple `HtmlInputText` component for accepting the user’s login name. This component has been assigned an identifier so that it can be referenced elsewhere (such as with the label described in 3).

- 5 This code creates another `HtmlInputText` component for the password, except we use the `<h:inputSecret>` tag, which renders as an HTML `<input>` tag of type “password”. This way, no one will be able to slyly stand over the user’s shoulder and see the password. This component has been assigned an identifier so that it can be referenced elsewhere, as in the `<h:outputLabel>` tag.
- 6 Submitting a form requires a `Command` component. This line declares an `Html-CommandButton` component, which generates an `ActionEvent` with the hardcoded outcome “success”. Hardcoding the action value tells JSF to bypass any application code and simply look for a matching outcome in the page’s navigation rule. Fortunately, in listing 8.2 we set up the navigation rule for this page so that the outcome “success” will always load `inbox.jsp`.

The output of the JSP is shown in listing 8.6.

Listing 8.6 The HTML output for the Login page with input components and a table layout

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
<head>
  <title>
    ProjectTrack
  </title>
</head>

<body>

<form id="_id1" method="post"
  action="/projects/faces/login.jsp"
  enctype="application/x-www-form-urlencoded">

<table cellpadding="0" cellspacing="0">
  <tr>
    <td>
      
    </td>
    <td>
      <table cellpadding="5" cellspacing="3">
        <tr>
          <td colspan="2">
            <font face="Arial, sans-serif" size="6">
              ProjectTrack
            </font>
          </td>
        </tr>
      </table>
    </td>
  </tr>
```

**Output of
HtmlForm**

```

<tr>
  <td>
    <label for="_id1:userNameInput">
      Enter your user name:
    </label>
  </td>
  <td>
    <input id="_id1:userNameInput"
      type="text"
      name="_id1:userNameInput"
      maxlength="30"
      size="20" />
  </td>
</tr>
<tr>
  <td>
    Password:
  </td>
  <td>
    <input id="_id1:passwordInput" type="password"
      name="_id1:passwordInput" value=""
      maxlength="20" size="20" />
  </tr>
</tr>
<tr>
  <td/>
  <td>
    <input type="submit" name="_id1:_id8" value="submit"
      title="Submit" />
  </tr>
</table>
</td>
</tr>
</table>

<input type="hidden" name="_id1" value="_id1" />
</form>

</body>
</html>

</form>

</body>
</html>

```

HtmlOutputText label

HtmlInputText for the username

HtmlInputSecret for the password

HtmlCommandButton

More output of HtmlForm

If you're wondering why `HtmlForm` has a `<hidden>` field at the end of the page, it's because the component processes that field in order to set its `submitted` property, which returns `true` if that field exists. This isn't something you usually have to worry about. Also note that the form posts back to itself, which, as we mentioned in the first part of this book, is called *postback*. This is the norm for JSF applications because the framework needs to associate events with the current view.

At this point, we have a working page. If you click the Submit button, you'll get an error page saying that `inbox.jsp` can't be found. (The obscurity of the message, however, will depend on your web container.) This is a good sign—it means that the navigation rule is configured correctly.

8.3 Sprucing things up

Okay, now we have a basic working Login page. But real applications are often a little nicer-looking than figure 8.4. The `` tag and plain-old form buttons are so '90s. Fortunately, you can use familiar HTML development tricks to make JSF applications look a little more modern (if you consider HTML modern, that is). What we're aiming for is something like figure 8.5, which uses JavaScript for a rollover effect on the button and Cascading Style Sheets (CSS) for formatting the title (instead of the `` tag).



Figure 8.5 The Login page with an image for a button, JavaScript for a rollover, and CSS for formatting.

8.3.1 Using an image for the button

An obvious way to spice things is to use a graphic for the button. This can be done by changing the `<h:commandButton>` tag as follows:

```
<h:commandButton action="success"
  image="#{facesContext.externalContext.requestContextPath}
  /images/submit.gif"/>
```

You may have noticed that the `image` property looks just like HTML. As a matter of fact, it is just passed through to the rendered HTML. Most of the components support these pass-through properties.

Pass-through properties do, however, support value-binding expressions, which is what we have in this example. The expression `"#{facesContext.externalContext.requestContextPath}"` references the application's context path, which is `"/projects"` in this case. This is necessary for `HtmlCommandButton` because the `image` property isn't automatically prefixed with the context path (as it is for `HtmlGraphicImage`). This is a feature that we hope will be added in a future version of JSF. In the meantime, you can either use a value-binding expression or use an `HtmlCommandLink` instead.

This expression references the `externalContext` property of `FacesContext`, which provides access to properties of the servlet or portlet environment. See chapter 11 for more information about this class.

By the way, this expression is also equivalent to the JSP expression `"<%= request.getContextPath() %>"`. To keep things consistent, it's usually better to use JSF expressions instead. We'll also use this expression to communicate the context path to JavaScript code.

8.3.2 Integrating with JavaScript

The pass-through HTML properties also allow us to integrate with client-side JavaScript quite easily. Although JSF components will often render JavaScript for you, it's quite likely that somewhere along the line you'll need to manually integrate with scripts, especially if you're converting an existing application.

For the Login page, we'll add a simple JavaScript function to create a standard "rollover" so our graphical button will appear to change the color of its text when a user places the mouse over it. We can do this by adding the following JavaScript to the `<head>` section of our page:

```
<script language="JavaScript">

  function set_image(button, img)
  {
```

```
        button.src = img;
    }

</script>
```

The code for the button itself must be changed as follows:

```
<h:commandButton action="success" title="Submit"
    image="#{facesContext.externalContext.
        requestContextPath}/images/submit.gif"
    onMouseOver="set_image(this,
        '#{facesContext.externalContext.requestContextPath}
        /images/submit_over.gif')"
    onMouseOut="set_image(this,
        '#{facesContext.externalContext.requestContextPath}
        /images/submit.gif');"/>
```

If you do a lot of front-end development, this sort of code should look pretty familiar to you. We use the standard client-side HTML event handlers to call our little function when the user places the mouse over the button (`onMouseOver`) or moves it away from the button (`onMouseOut`). The function receives a reference to the button element (which will be rendered as a standard `<input>` element of type “image”) by using this identifier. It also receives a string with the name of the graphic we’d like the browser to display for that button. In the HTML 4 document object model (DOM), this element has an `src` property, which is what we set to equal the new image name in the `set_image()` function. The graphic `submit_over.gif` looks exactly like `submit.gif`, except that the color of the text is black. Consequently, while the user’s mouse is over the button, the text color appears to change to black; it turns back to green when the user moves the mouse away.

Note that we’ve prepended the image URL with the value-binding expression for the web application’s context path, which is “/projects” in this case. This way, we ensure that the images use an absolute URL instead of a relative URL. This allows us to change the context root without worrying about hardcoded absolute paths.

This simple example demonstrates the fact that integrating JavaScript into JSF applications is as easy as integrating JavaScript into standard JSP applications. Generally speaking, if the custom tag you’re using for the component has an HTML property, you can be sure it’s going to be in the HTML rendering of the component. This allows you to attach JavaScript event handlers to any component tag that accepts the proper properties. You can also put script elements and references in JSF pages just as you can with any other JSP page.

Client-side tasks like rollovers are extremely common and often require lots of little bits of JavaScript code—sometimes way more than should be necessary. This makes them perfect candidates for JSF components that handle all of the

JavaScript for you. In this example, it'd be much nicer to just specify both image filenames as properties of the `<h:commandButton>` tag and forget about all of the event handling and JavaScript. We develop a renderer with this functionality in online extension chapter 17.

Adding an image to the button makes things look a little better, but we could use some additional formatting as well.

8.3.3 Adding Cascading Style Sheets

Just as we added JavaScript by using pass-through HTML properties, we can also add Cascading Style Sheets (CSS). In the previous listings, we formatted the text “ProjectTrack” using the HTML `` element. Most sites these days use CSS for formatting HTML elements, and we have been using this approach throughout the book.

Some IDEs will give you tools for creating CSS styles without any knowledge of CSS (see figure 4.1 for a screen shot), but most web developers are quite familiar with it already. Let's start by creating a file, `stylesheet.css`, and then place it in the root of the web application directory. The first style will be the one used to format the name of the application (replacing the `` tag):

```
.login-heading
{
  font-family: Arial, sans-serif;
  font-size: 26pt;
}
```

Next we can add the style sheet reference to our Login page:

```
<head>
...
  <link rel="stylesheet" type="text/css"
        href="stylesheet.css"/>
...
</head>
```

Now we can replace the `` tags:

```
<h:outputText value="ProjectTrack" styleClass="login-heading"/>
```

Most of the components display the `class` property as an HTML `` element. The previous code fragment renders to the following HTML:

```
<span class="login-heading">ProjectTrack</span>
```

That's all there is to it. Every tag in the standard HTML tag library that supports style sheets has a special property called `styleClass`. Panels and `HtmlDataTable` are a little more complicated (they have styles for different rows and columns) but any property ending in "Class" is used to reference a CSS style.

We've now achieved the more attractive page shown in figure 8.5. But a page alone isn't all that exciting.

8.4 Adding validators

Even though our page works as is, it's not being used as much more than a standard web form with no logic. No matter what you type in the input fields, clicking the Submit button always forwards you to the `inbox.jsp` page. Because JSF has built-in support for validation, it makes sense to add validators so that the interface feels a little more interactive to users. Adding validation during UI prototyping also helps users and developers get a shared vision of how the application should behave.

In chapter 4, we showed you how to use `HtmlMessage` and `HtmlMessages` to display validator error messages. However, our page design has no place for reporting such errors. We need to report errors for both fields, which means adding another column to the nested table. The revised mock-up of the page is shown in figure 8.6. You can see that we've included a new column that is used to display error messages—one cell for each input field. If no errors are present, the column should be empty (in other words, the page should look like the original mock-up in figure 8.1, page 289).

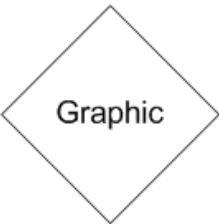
	ProjectTrack		
	Enter name:	<input type="text"/>	Input error msg
	Password:	<input type="password"/>	Input error msg
	<input type="submit" value="Submit"/>		

Figure 8.6 A mock-up of the Login page with a column for errors.

These error messages should grab the user's attention, so we'll add a new CSS class to our style sheet for them:

```
.errors {
  color: red;
}
```

No rocket science here; this just ensures that our error message show up in red.

Now that we know where to put the error messages and what they'll look like, let's take a look at the requirements for these two fields. Because both the username and password are necessary to validate the user, both fields are required. They must be at least five characters long. The maximum length of the name field is 30; the maximum length of the password field is 15. This translates to setting the `required` property to `true`, and adding a `Length` validator, respectively. The relevant portions of the code, changed to add validation, is shown in listing 8.7.

Listing 8.7 Adding validation to the Login page

```
...
<table cellpadding="0" cellspacing="0">
  <tr>
    <td>
      <h:graphicImage url="/images/logo.gif" alt="Welcome to ProjectTrack"
        title="Welcome to ProjectTrack" width="149"
        height="160"/>
    </td>
    <td>
      <table cellpadding="5" cellspacing="3">
        <tr>
          <td colspan="3">
            <h:outputText value="ProjectTrack"
              styleClass="login-heading"/>
          </td>
        </tr>
        <tr>
          <td>
            <h:outputLabel for="userNameInput">
              <h:outputText value="Enter your user name:"/>
            </h:outputLabel>
          </td>
          <td>
            <h:inputText id="userNameInput" size="20" maxLength="30"
              required="true">
              <f:validateLength minimum="5"
                maximum="30"/>
            </h:inputText>
          </td>
        </tr>
      </table>
    </td>
  </tr>
</table>
```

Additional column for errors

Require input

Length validator

```

        </td>
        <td>
            <h:message for="userNameInput"
                styleClass="errors" />
        </td>
    </tr>
    <tr>
        <td>
            <h:outputLabel for="passwordInput">
                <h:outputText value="Password:" />
            </h:outputLabel>
        </td>
        <td>
            <h:inputSecret id="passwordInput" size="20" maxLength="20"
                required="true">
                <f:validateLength minimum="5" maximum="15" />
            </h:inputSecret>
        </td>
        <td>
            <h:message for="passwordInput" styleClass="errors" />
        </td>
    </tr>
    <tr>
        <td/>
        <td>
            <h:commandButton action="success" title="Submit"
                image="#{facesContext.externalContext.requestContextPath}
                    /images/submit.gif"
                onMouseOver="set_image(this,
                    '#{facesContext.externalContext.requestContextPath}
                    /images/submit_over.gif') "
                onMouseOut="set_image(this,
                    '#{facesContext.externalContext.requestContextPath}
                    /images/submit.gif');"/>
        </td>
    </tr>
    </table>
</td>
</tr>
</table>
...

```

HtmlMessage
component

Now that we've added validators, the `HtmlMessage` components on the page will display error messages if there are any, as shown in figure 8.7.

With validators on the page, JSF now will automatically handle validation, remembering the value of the input controls even if the value is incorrect. The actual text of the error messages could use some help, so let's tackle that task now.

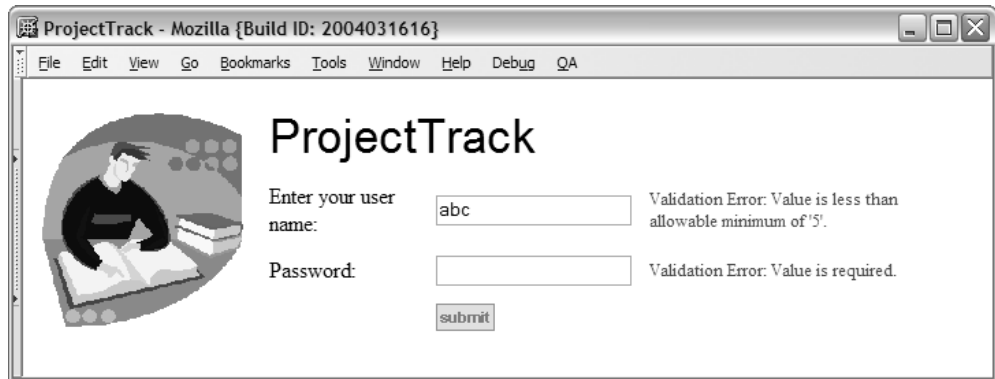


Figure 8.7 The Login page with validation errors.

8.4.1 Customizing validation messages

The validation error messages shown in figure 8.7 are the standard messages that ship with Sun's JSF RI [Sun, JSF RI]. Other implementations may have nicer messages, but because this one doesn't, we'll make some changes.

In the directory listing shown in figure 8.1, we had a simple file called `ptrack-Resources.properties` in the `WEB-INF/classes` directory. This is a resource bundle—a file is for customizing error messages and also localizing other application strings. During prototyping, we can use it to make our validation errors look a little more user-friendly. In part 3, we'll use resource bundles for localization as well.

The file `ptrackResources.properties` is a standard Java properties file, which has just plain text and name/value pairs. As long as we use the proper validation message key, we can change the value. Listing 8.8 shows the changes necessary for the two validation messages used in the login page.

Listing 8.8 Customized validation error messages

```
javax.faces.component.UIInput.REQUIRED=This field is required.
javax.faces.component.UIInput.REQUIRED_detail=
    Please fill in this field.
javax.faces.validator.LengthValidator.MAXIMUM=
    This field must be less than {0} characters long.
javax.faces.validator.LengthValidator.MINIMUM=
    This field must be at least {0} characters long.
```

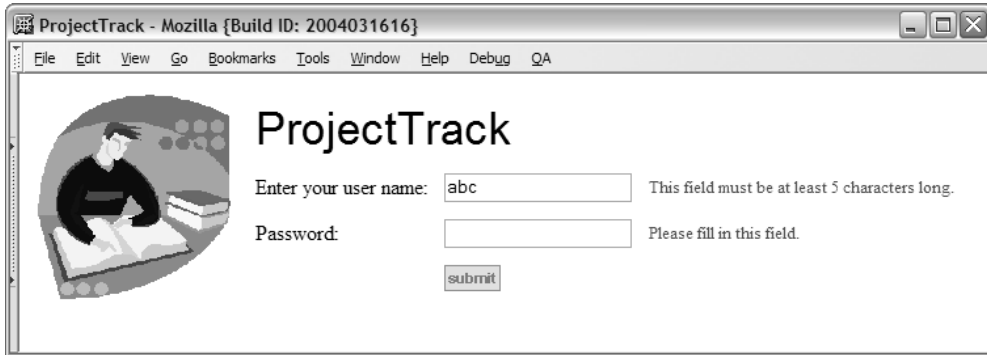


Figure 8.8 The login page with customized validation errors.

This customizes the summary and detail text for input controls with the `required` property set to `true`, and the summary text for the `Length` validator (no detail text has been specified).

All we have to do now is tell JSF about this file so that it will use it for validation error messages. This can be done by adding an `<application>` node to `faces-config.xml`:

```
<application>
  <message-bundle>ptrackResources</message-bundle>
</application>
```

Note that we don't use the extension ".properties" here—JSF will automatically append it as necessary. After making this change (and restarting the application), we get the friendlier messages shown in figure 8.8, which make the page a little more like a real application. For more information on validation and customizing error messages, see chapter 6.

The page is now pretty complete, but we can still play with the layout.

8.5 Improving layout with `HtmlPanelGrid`

You may have noticed that even though our page has a decent amount of JSF components, the layout is still handled by standard HTML tables. Another approach is to use multiple `HtmlPanelGrid`s for layout. This component is the closest thing JSF has to Swing's layout managers (as far as standard components go).

Using `HtmlPanelGrid` allows you to work more exclusively with JSF components, which means you can spend less time dealing with the nuances of HTML. In other words, you won't waste time trying to figure out where that missing `<td>`

element went. Well-developed components are also intimately familiar with HTML standards and can change their output depending on the browser type, which saves you the effort of doing so yourself.

So, let's modify our page. The goal is to duplicate the structure shown in figure 8.6: a main two-column table with the image on the left, and a nested table on the right. The nested table will have a heading with the text "ProjectTrack", followed by rows that have input fields, buttons, and output components.

`HtmlPanelGrid` is the perfect choice for this because it lays out components in tables without the need for a value-binding expression. The component displays an HTML table, so just as we had one HTML table nested inside another, we'll have one `HtmlPanelGrid` nested inside another. Our completed login page, revised to use panels, is shown in listing 8.9.

Listing 8.9 The JSP source for our completed Login page using an `HtmlPanelGrid` for component layout

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

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

<f:view>
<html>
  <head>
    <title>
      <h:outputText value="ProjectTrack"/>
    </title>

    <link rel="stylesheet" type="text/css"
          href="/projects/stylesheet.css"/>

    <script language="JavaScript">

      function set_image(button, img)
      {
        button.src = "/projects" + img;
      }

    </script>
  </head>

  <body>

  <h:form>

    <h:panelGrid columns="2" border="0" cellpadding="3"
                 cellspacing="3">
```

1

**HtmlPanelGrid
instead of table**

```
<h:graphicImage url="/images/logo.gif"
  alt="Welcome to ProjectTrack"
  title="Welcome to ProjectTrack"
  width="149" height="160" />
```

```
<h:panelGrid columns="3" border="0"
  cellpadding="5"
  cellspacing="3"
  headerClass="login-heading">
```

2 Nested HtmlPanelGrid
for input components

```
<f:facet name="header">
  <h:outputText value="ProjectTrack" />
</f:facet>
```

3 Header
facet

```
<h:outputLabel for="userNameInput">
  <h:outputText value="Enter your user name:" />
</h:outputLabel>
<h:inputText id="userNameInput" size="20" maxLength="30"
  required="true">
  <f:validateLength minimum="5" maximum="30" />
</h:inputText>
<h:message for="userNameInput" />

<h:outputLabel for="passwordInput">
  <h:outputText value="Password:" />
</h:outputLabel>
<h:inputSecret id="passwordInput" size="20" maxLength="20"
  required="true">
  <f:validateLength minimum="5" maximum="15" />
</h:inputSecret>
<h:message for="passwordInput" />
```

```
<h:panelGroup/> ← 4 HtmlPanelGroup placeholder
<h:commandButton action="success" title="Submit"
  image="#{facesContext.externalContext.requestContextPath}
  /images/submit.gif"
  onMouseOver="set_image(this,
  '#{facesContext.externalContext.requestContextPath}
  /images/submit_over.gif')"
  onMouseOut="set_image(this,
  '#{facesContext.externalContext.requestContextPath}
  /images/submit.gif');"/>
<h:panelGroup/>
```

```
</h:panelGrid>
```

```
</h:panelGrid>
</h:form>

</body>
```

```
</html>  
</f:view>
```

- 1 First, we add an enclosing `HtmlPanelGrid`. This component uses the `columns` property to determine the number of columns to display. It works by rendering its child components in order, based on the number of columns. Because two columns have been specified it will display two child components in each row (one per column). It will display them in order: the first component will be displayed in column one, row one. The second will be displayed in row one, column two. The third will be displayed in row two, column one. The fourth will be displayed in row two, column two, and so on.

We've specified only two child components: an `HtmlGraphicImage` and another `HtmlPanelGrid`, which is used to lay out additional controls. Consequently, this will render as a table with a single row and two columns; the left column will display an image tag, and the right will have a nested table rendered by the nested `HtmlPanelGrid`.

- 2 This nested `HtmlPanelGrid` will lay out all of the other components on the form. The `headerClass` property specifies the CSS style for the header row. The `columns` property is set to 3, so every group of three child components will make up a single row (one per column). There are three groups: one for entering the name, one for entering the password, and one for the Submit button.
- 3 In order to display the text "ProjectTrack" as a single header row with a single column, we can place an `HtmlOutputText` inside of the header facet. The `headerClass` property of the parent `HtmlPanelGrid` will be used to style it appropriately.
- 4 If you look at figure 8.6, the bottom row of the nested table has two empty cells (the first and last). In our HTML table, we achieved this effect with the infamous empty cell (`<td/>`). When using `HtmlPanelGrid`, we must do the same thing to ensure that Faces places the button in the middle column. This can be done with the `HtmlPanelGroup`, which is a panel with no visual representation. If we left this out, the component would place the next component (an `HtmlCommandButton`) underneath the password input label. This isn't the desired behavior—we want the buttons to be under the password input field itself.

We've now successfully modified the page to use panels for layout. It looks exactly as it did in figure 8.5, because the table structure rendered by the two `HtmlPanelGrid` components is the same. The generated HTML is shown in listing 8.10

Listing 8.10 The Login page HTML output with layout generated by two HtmlPanelGrids

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
<head>
  <title>ProjectTrack</title>
  <link rel="stylesheet" type="text/css" href="stylesheet.css">
  <script language="JavaScript" type="text/javascript">

      function set_image(button, img)
      {
        button.src = img;
      }

  </script>
</head>

<body>
  <form id="_id1" method="post"
    action="/projects/faces/login.jsp"
    enctype="application/x-www-form-urlencoded">
    <table border="0" cellpadding="3"
      cellspacing="3">
      <tbody>
        <tr>
          <td>
            
          </td>
          <td>
            <table border="0"
              cellpadding="5"
              cellspacing="3">
              <thead>
                <tr>
                  <th class="login-heading"
                    colspan="3"
                    scope="colgroup">
                    ProjectTrack
                  </th>
                </tr>
              </thead>
              <tbody>
                <tr>
                  <td>
                    <label for="_id1:userNameInput">

```

① Main
HtmlPanelGrid

② Nested
HtmlPanelGrid

③ Header
facet

```
        Enter your user name:
    </label>
</td>

<td>
    <input id="_id1:userNameInput" type="text"
           name="_id1:userNameInput" maxlength="30"
           size="20">
</td>

<td></td> ← 4 HtmlPanelGroup
</tr>
<tr>
<td>
    <label for="_id1:passwordInput">
        Password:
    </label>
</td>

<td>
    <input id="_id1:passwordInput"
           type="password" name="_id1:passwordInput"
           value=""
           maxlength="20" size="20">
</td>

<td></td>
</tr>
<tr>
<td></td>

<td>
    <input type="image"
           src="/projects/images/submit.gif"
           name="_id1:_id13"
           onMouseOut="set_image(this,
                                   '/projects/images/submit.gif');"
           onMouseOver="set_image(this,
                                   '/projects/images/submit_over.gif');"
           title="Submit">
</td>

<td></td>
</tr>
</tbody>
</table>
</td>
```

```
        </tr>
      </tbody>
    </table>

    <input type="hidden" name="_id1" value="_id1">
  </form>
</body>
</html>
```

- ❶ This is the primary table, rendered by the first `<h:panelGrid>` tag. You can see that the HTML pass-through properties were indeed passed through. You may also have noticed that the order of the properties is different than the order in the original JSP code. Components don't necessarily guarantee the order of pass-through properties; they just guarantee the content.
- ❷ This is the nested table, rendered by the second `<h:panelGrid>` tag.
- ❸ This is the nested `HtmlPanelGrid`'s header facet. You can see that it's a single row that spans all three columns, and the CSS class is the same as the parent `HtmlPanelGrid`'s `headerClass` property.
- ❹ `HtmlPanelGroup` doesn't have a visual representation, so it just outputs empty cells.

We now have a Login page, complete with JavaScript, CSS, validation, custom error messages, and a panel-based layout. We covered each and every step of building this page for a good reason—this way, we can skip the topics explained here when we describe the other pages, so you can focus on the unique aspects of the other pages.

8.6 Summary

In this chapter, we built a static Login page, step by step, with JSF and JSP. We began by examining each element of the Login page, one by one. The first step was to create the page, importing the proper tag libraries and adding `HtmlGraphicImage` and `HtmlOutputText` components. We then added a form for collecting the username and password. Next, we spiced up things up with a button image, Cascading Style Sheets, and a little bit of JavaScript for an image rollover. (Bear in mind that JSF will typically limit the amount of required JavaScript, because components should generate it themselves.)

We then added validators to the Login page, making sure we had enough room for displaying the error messages. This is key—if you add validation and you want to redisplay the page with errors, you must allow space for those errors to be displayed. Next, we customized those error messages with a resource bundle.

Finally, we demonstrated a powerful technique: laying out components with panels as opposed to HTML tables. This technique lets you focus on the conceptual view of the layout as opposed to the specifics of HTML.

In the next chapter, we'll build the rest of the UI using the techniques learned in this chapter.

To order this book, go to www.manning.com/mann
or visit your favorite bookstore.

JAVASERVER FACES IN ACTION

Kito D. Mann • foreword by Ed Burns

JavaServer Faces helps streamline your web development through the use of UI components and events (instead of HTTP requests and responses). JSF components—buttons, text boxes, checkboxes, data grids, etc.—live between user requests, which eliminates the hassle of maintaining state. JSF also synchronizes user input with application objects, automating another tedious aspect of web development.

JavaServer Faces in Action is an introduction, a tutorial, and a handy reference. With the help of many examples, the book explains what JSF is, how it works, and how it relates to other frameworks and technologies like Struts, Servlets, Portlets, JSP, and JSTL. It provides detailed coverage of standard components, renderers, converters, and validators, and how to use them to create solid applications. This book will help you start building JSF solutions today.

What's Inside

How to

- Use JSF widgets
- Integrate with Struts and existing apps
- Benefit from JSF tools by Oracle, IBM, and Sun
- Build custom components and renderers
- Build converters and validators
- Put it all together in a JSF application

An independent enterprise architect and developer, **Kito D. Mann** runs the JSFCentral.com community site and is a member of the JSF 1.2 and JSP 2.1 Expert Groups. He lives in Stamford, Connecticut with his wife, two parrots, and four cats.

To order this book, go to www.manning.com/mann or visit your favorite bookstore.

ONLINE BONUS!

Exclusive to owners of this book: free online access to over 300 **additional** pages of substantial content. That's a total of over 1,000 pages of *JavaServer Faces in Action*!

“Can’t wait to make it available to the people I teach.”

—Sang Shin, Java Technology Evangelist,
Sun Microsystems Inc.

“This book unlocks the full power of JSF... It’s a necessity.”

—Jonas Jacobi
Senior Product Manager, Oracle

“... explains advanced topics in detail. Well-written and a quick read.”

—Matthew Schmidt, Director,
Advanced Technology, Javalobby

“... by a programmer who knows what programmers need.”

—Alex Kolundzija, Columbia House

“A great reference and tutorial!”

—Mike Nash, JSF Expert Group Member,
Author, *Explorer’s Guide to Java Open Source Tools*



Ask the Author



Ebook edition

www.manning.com/mann



ISBN 1-932394-12-5



MANNING

\$49.95 US/\$74.95 Canada