


BEYOND THE BOOK

2

Topics in This Supplement

- 
- Component Style Attributes
 - Cascading Style Sheets
 - Page Fragments
 - Template Items
 - Visual Component Linking
 - Session Timeout
 - Navigation with Wildcard Expressions

Appendix

B

Welcome to Beyond the Book 2. In this document, we continue to step through examples that illustrate aspects of Creator not covered in the first edition of Java Studio Creator Field Guide. In particular, we concentrate on ways you can use Creator to build web applications that have a unified look and feel using style sheets and page fragment components.

B.1 About Style

Web page designers have a number of choices for specifying the look of a web page using Creator. These choices include specifying individual style attributes (such as color or font size characteristics, for example) for a component, as well as other “pass-through” HTML attributes, such as border and cellpadding that apply to table-type components. You specify these attributes by modifying the component’s property sheet directly. This gives you control over the look of a specific component. The disadvantage, however, is that you must specify attributes for each component manually by editing its property sheet. This quickly becomes tedious. Furthermore, it becomes difficult to impose a uniform look to a web page.

Creator offers several features to help designers streamline web page design to achieve a consistent look. They are as follows.

- Using style sheets to specify global styles;
- Using Page Fragment Box components for specifying page building blocks; and
- Using Creator templates as a starting point for new web applications or items.

Let's start with style sheets.

Cascading Style Sheets

Creator uses Cascading Style Sheets (currently CSS 2.1) to control the look of JSF standard components. CSS is a standard that allows a web designer to specify style characteristics. The style characteristics apply to a document in a *cascading* fashion: that is, a style applies to a given level and subsequent styles can in turn apply on top of these "inherited" styles. If you don't specify a property for an element, it inherits the property from its "parent." For example, you can specify that all text in a document is (color) navy. You can then specify that text in a footer is a smaller text size. The footer text will be *both* navy and the smaller size since the footer-specific style inherits all properties specified for the global style.

You can read more about Cascading Style Sheets at the Cascading Styles Home Page: <http://www.w3.org/Style/CSS/>. The web site also includes tutorials about style sheets.

Using Attribute style

All JSF standard components have at least two attributes (and some components have additional attributes) that take style characteristics: `style` and `styleClass`. Attribute `style` accepts style declarations in the form

```
property1: value1; property2: value2; . . . propertyN: valueN
```

Let's look at an example.

1. In Creator's Welcome Page, click Create New Project. Specify the project name as **SimpleWebApp**. The selected project type is J2EE Web Application. Click OK. Creator creates a project for you and brings up the design canvas.
2. Click anywhere in the design canvas. In the Properties window, set the page's `Title` attribute to **Simple**.
3. From the JSF Standard Components palette, select component Output Text and drop it onto the design canvas.
4. Make sure the component is selected and type the text **My Title** followed by `<Enter>`. This sets the `value` attribute; Creator displays this text on the page.

5. Again, make sure the component is still selected and examine the Properties window. Under Appearance, click the small editing box opposite attribute `style`. Creator pops up an editing dialog and displays the component's style attribute. Depending where on the page you placed the output text component, you should see a style declaration similar to the following.

```
left: 48px; top: 24px; position: absolute
```

This specifies property `left` with value `48px`, property `top` with value `24px`, and property `position` with value `absolute`. Note that the final value does not require a terminating semi-colon.

You use a component's `style` attribute to control its appearance. For example, suppose you'd like to make the text in this component blue, larger, and bold. Use the `color` property to control the text's color, the `font-size` property¹ to adjust its size, and the `font-weight` property to adjust its weight or thickness. Follow these steps.

1. The style editing dialog should still be open. To the property declarations that are already listed above, add the following (note that you need a semi-colon after value `absolute` before you can add more property-value pairs).

```
; color: blue; font-size: 200%; font-weight: bold
```

2. Click OK. The output text should now appear with its new style characteristics.
3. Now select the tab labeled Source at the bottom of the editing pane. Creator displays the page's JSP source. You'll see the output text component in the source, as shown. The style declaration is bold.

```
<h:outputText binding="#{Page1.outputText1}" id="outputText1"  
  style="left: 48px; top: 24px; position: absolute;  
  color: blue; font-size: 200%; font-weight: bold"  
  value="My Title"/>
```

4. Go ahead and deploy the application. Select Build > Run Project (or click the green chevron in the toolbar). The page with the text "My Title" appears in

1. In general, it is better to use *relative* values for property `font-size` instead of absolute sizes. This allows users to see text in correct proportions even if they have customized their browsers to use unusual font size settings.

your browser with the style characteristics you specified as shown in Figure B-1.



Figure B-1 Setting attribute `style` to control text characteristics

Using Attribute `styleClass`

A web designer's task can quickly become tedious if each component in a large web application must have its style specified individually. To address this, all JSF standard components include attribute `styleClass`, which is a comma separated list of style classes. You define and store a style class in a text file called a *style sheet*. Creator provides a default style sheet, `stylesheet.css`, that is included in each project you create. When you add style classes to the style sheet, you can then reference them in the component's `styleClass` attribute.

Let's examine the default style sheet for Creator projects. First, copy the **SimpleWebApp** project to a new project called **SimpleWebApp2**.

1. If it's not already opened, open project **SimpleWebApp** that you created in the previous section.
2. From the File menu, select Save Project As and provide the name **SimpleWebApp2**. You'll make changes to the **SimpleWebApp2** project.
3. Click anywhere in the design canvas of the **SimpleWebApp2** project. In the Properties window, change the page's `Title` attribute to **Simple 2**.
4. In the Project Navigator window, expand folder Resources.
5. Under Resources, double-click `stylesheet.css`. Creator brings up this file in the editor pane.

A style sheet is a collection of style *rules*. Each rule consists of a *selector* and a *declarator*. The selector identifies an HTML element or style class name to which the rule applies. The curly braces encompass the declarator, which is the semi-colon separated list of property-value pairs. While the component's `style` attribute lists the property-value pairs for a given component, a rule is a collection of property-value pairs that is named.

As an example, you see that the first rule applies to selector HTML element *body*, as follows.

```
body {  
  background-color: white;  
  color: black  
}
```

Here we have two property-value pairs: property `background-color` has value `white` and property `color` has value `black`. This rule, then, applies to all HTML `<body>` elements, as well as any elements declared inside of `<body>`. Thus, nested (“children”) elements inherit the property-value settings from their enclosing (“parent”) elements. The `body` style rule is a good place to list global settings for your web application, such as basic font characteristics, text color, and background color.

Now return to the editing pane for **stylesheet.css**. You’ll see additional rules. Rules that contain an initial dot are *style classes*. Once you define them in the style sheet for your project, you can specify the class names in a component's `styleClass` attribute. One group of rules (`.list-header`, `.list-paging-header`, `.list-paging-footer`, `.list-row-even`, and `.list-row-odd`) applies to the JSF data table component. When you select a data table component for your page, Creator automatically applies these style classes to the data table's various style class attributes. (We’ll examine the data table component later in the chapter). The second set of rules (`.infoMessage`, `.warnMessage`, `.errorMessage`, `.fatalMessage`) applies to JSF inline message and message list components. When you place one of these components on your page, Creator automatically assigns these style classes to the various style class attributes.

Let's use the output text component's `styleClass` attribute to modify its look. To do this, we're going to define a new style rule in file **stylesheet.css**.

1. Make sure that file **stylesheet.css** is in the editing pane. (Click the tab labeled **stylesheet.css** at the top of the pane if the file is not currently showing.)

2. Scroll to the top of the file. Add the following style rule after the style rule for `body`. Copy and paste from your Creator book's `FieldGuide/Examples/Beyond2/snippets/Simple_headingStyle1.txt`.

```
.headingStyle {
  color: green;
  font-size: 200%;
  font-weight: bold;
  font-style: italic
}
```

3. Write out the file by selecting `File > Save All` (or click the double floppy icon on the toolbar).

Now you'll modify the first output text component and add a second output text component to use the style rule you just created.

1. Return to the `Page1.jsp` design canvas by selecting tab labeled `Page1.jsp` at the top of the editing pane.
2. Select the output text component entitled `My Title`.
3. In the Properties window, remove the attribute `style` property-value pairs you added previously that affected `color`, `font-weight`, and `font-size`.
4. Opposite attribute `styleClass`, enter the text `headingStyle` followed by `<Enter>`. (Note that you do not include the initial dot from the style class name.)



Creator Tip

In order for the style changes to be reflected in the design canvas, you must save any editing changes made to the style sheet. Use `File > Save All` from the main menu.

5. From the JSF standard components palette, select `Output Text` and drop it on the canvas below the first component.
6. While it's still selected, type in the text `Second Title` followed by `<Enter>`. (This sets the `value` attribute.)
7. In the Properties window opposite attribute `styleClass`, enter the text `headingStyle` followed by `<Enter>`. (Again, do not include the initial dot from the style class name.)
8. Deploy and run the project by clicking the green chevron on the toolbar. In the browser, you'll see two headings on the page. Note that both headings have the same style, as shown in Figure B-2.

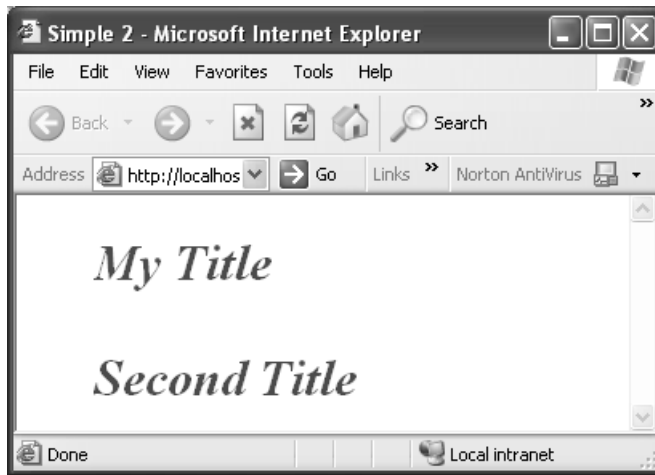


Figure B-2 Setting attribute `styleClass` to control text characteristics

The advantage of using a style class over specifying the style attribute for each component is that, once you develop a style rule (or set of rules), you can apply them to the components on your page. Then, if you want to modify the look, you only need to modify the style rule without changing any of the components' attributes.

Of course, once you develop a set of style rules that you like, you can create a style sheet *template* that can be brought into your projects. Let's do this now.

Creating a Style Sheet Template

For this example, we're going to modify the existing `stylesheet.css` and create a template for this item.

1. If it's not already opened, open project **SimpleWebApp2** that you created previously and click anywhere inside the design canvas.
2. From the File menu, select Save Project As and provide the name **SimpleWebApp3**. You'll make changes to the **SimpleWebApp3** project.
3. Click anywhere in the design canvas of the **SimpleWebApp3** project. In the Properties window, change the page's `Title` attribute to **Simple 3**.
4. Click the tab labeled `stylesheet.css` at the top of the editor pane to display the default style sheet in the editor.
5. Replace the rules for `body` and `.headingStyle` as shown. Copy and paste from your Creator book's **FieldGuide/Examples/Beyond2/snippets/**

Appendix B Beyond the Book 2

Simple_headingStyle2.txt. (Only replace the rules for `body` and `.headingStyle`. Make sure that the other rules are unchanged.)

```
/* custom style rules */

body {
    background-color: #eeeeee;
    color: navy;
    font-family: Verdana, Helvetica, sans-serif;
}

.headingStyle {
    font-size: 150%;
}
```

6. Save the project (click File > Save All from the top menu or click the double floppy icon from the toolbar).
7. Now click the tab labeled **Page1.jsp** at the top of the editor pane to display the web page in the design canvas.
8. Deploy and run the application. You'll see that the page's looks have changed, as shown in Figure B-3.



Figure B-3 Modifying stylesheet.css

Note that the style rule for `<body>` globally changes the font characteristics of this web application. In addition to the output text components already on the page, any component that uses fonts will inherit the style specified for `<body>` for its text. For example, if you include a button, link action, or text

field component, each of these will also use navy text and the Verdana font family.

Additionally, the style class `.headingStyle` specifies a font size that is 150% larger than the standard font size. Since both the output text components have attribute `styleClass` set to `headingStyle`, the text is further modified from the styles we specified for `body` to reflect the larger size.

As a final step, let's create a *template* for this style sheet.

1. In the Project Navigator window for this project (**SimpleWebApp3**), expand folder Resources and select file **stylesheet.css**.
2. Right-click **stylesheet.css** and select menu item Save as Template.
3. Creator pops up the Save as Template dialog.
4. Under Select the category under which the new template will appear, select Resources. Click OK.

This saves the style sheet as a template under a default name, **stylesheet_1.css**. Let's rename it.

1. From the main menu bar, select Tools > Options. Creator pops up the Options dialog.
2. Select the radio button Advanced.
3. Scroll down to Source Creation and Management and expand folders Templates, then Resources.
4. Under Resources, right-click file **stylesheet_1.css** and select Rename from the context menu, as shown in Figure B-4.
5. Enter the new file name **MyCompanyStyleSheet** and click OK.
6. Click Close to close the Options dialog.
7. Save and Close project **SimpleWebApp3**.

Using a Style Sheet Template

Once you create a style sheet template, you can use it in a project. To show you this, let's return to project **SimpleWebApp2** that you created previously.

1. From the Welcome page, click Open Existing Project, select project **SimpleWebApp2** from the list, and click Open (or select project **SimpleWebApp2** from the list of recent projects).
2. Bring up **Page1.jsp** in the design canvas by selecting the tab labeled **Page1.jsp** at the top of the editor pane, if it's not already displayed. Note that the two output text components use style class `headingStyle` defined in **stylesheet.css**.
3. Open the **MyCompanyStyleSheet** template you created previously. From the top menu, select Tools > Options. Make sure the Advanced radio button is selected.

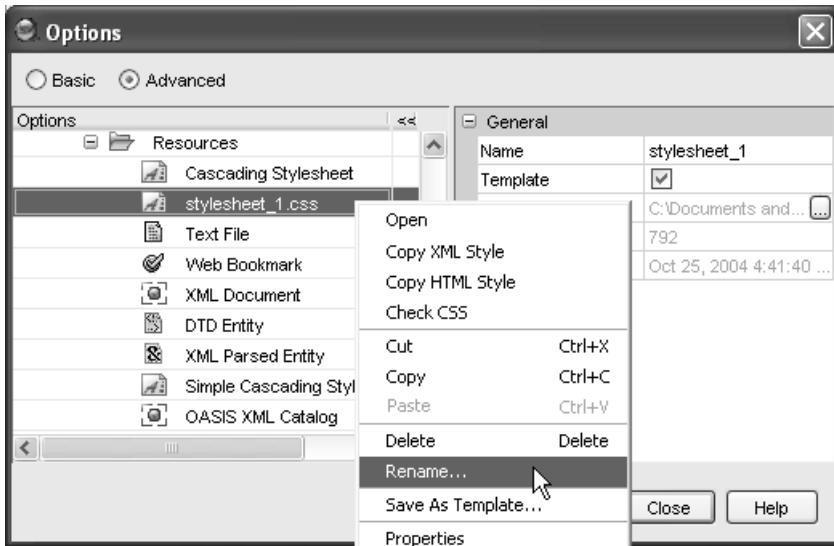


Figure B-4 Renaming the style sheet template

4. Scroll down to Source Creation and Management and expand folders Templates, then Resources.
5. Under Resources, right-click file **MyCompanyStyleSheet.css** and select Open from the context menu. Creator brings up the style sheet in the editor pane. Click Close to close the Options dialog.
6. In the editor, select the entire file and type **<Ctrl-C>** to copy its contents (or use Edit > Copy).
7. Now open the default style sheet, **stylesheet.css**, in the editor pane. Select the tab labeled **stylesheet.css** or, from the Project Navigator window under Resources, double-click file **stylesheet.css**.
8. In the editor, select the entire file and type **<Ctrl-V>** to replace the default contents (or use Edit > Paste). Save the changes (use File > Save All or click the double floppy icon on the toolbar).

Now if you return to the **Page1.jsp** web page, you'll see the styles from the saved style sheet template reflected on the design canvas.

About Template Items

When you save an item as a template, such as the **stylesheet.css** file in the previous section, Creator saves this file in your personal Creator directory (under

`.Creator`) used for customizing the system. The template, then, is not connected to a project, but it lives in this directory.

To find the template file `MyCompanyStyleSheet.css`, expand folder `.Creator > 1_0 > system > Templates > Other`.

Creator Tip

*Here's how to determine your customized Creator directory. From the main menu, select `Help > About Java Studio Creator`. When the window pops up, select the tab labeled `Detail`. The user directory appears after the label **User Dir**.*



B.2 About Page Fragments

Another valuable tool for the web designer is the page fragment component, providing a way to define a building block for a web application page. You can place components inside a page fragment and then use this page fragment with subsequent pages. Typically, you use page fragments to hold parts of your web page that you'd like to standardized for a uniform look, such as images used as page headers, standard menus, or even footers that contain copyright notices (for example).

A page fragment is a useful mechanism for reuse, but it does have some caveats. It is inserted inline into its containing document on the server. This means that the page fragment can only contain elements that are valid at the point of inclusion. As you work through the example project, you'll notice that page fragments are embedded in a `<div>` element (generated by Creator), and they do not contain elements such as `<head>` or `<body>`, which already exist in the containing page.

Using page fragments involves two steps: first you must create the page fragment, then you can place it on a page. You can also save the page fragment as a template to facilitate reuse by other projects. As an example, let's build a project. It consists of three pages: a Home (login) page, a Courses page, and a Books page. Figure B-5 shows the basic layout of the Home page. The header is a page fragment that contains an image component, the footer is a page fragment that contains an output text component, and the left menu is a page fragment that consists of a grid panel component holding navigation links.

This web application requires a user to issue a login first name and last name. The names are stored in session scope and, in this application, are used to adorn the navigation menu on the left. The user must login before proceeding to subsequent pages.

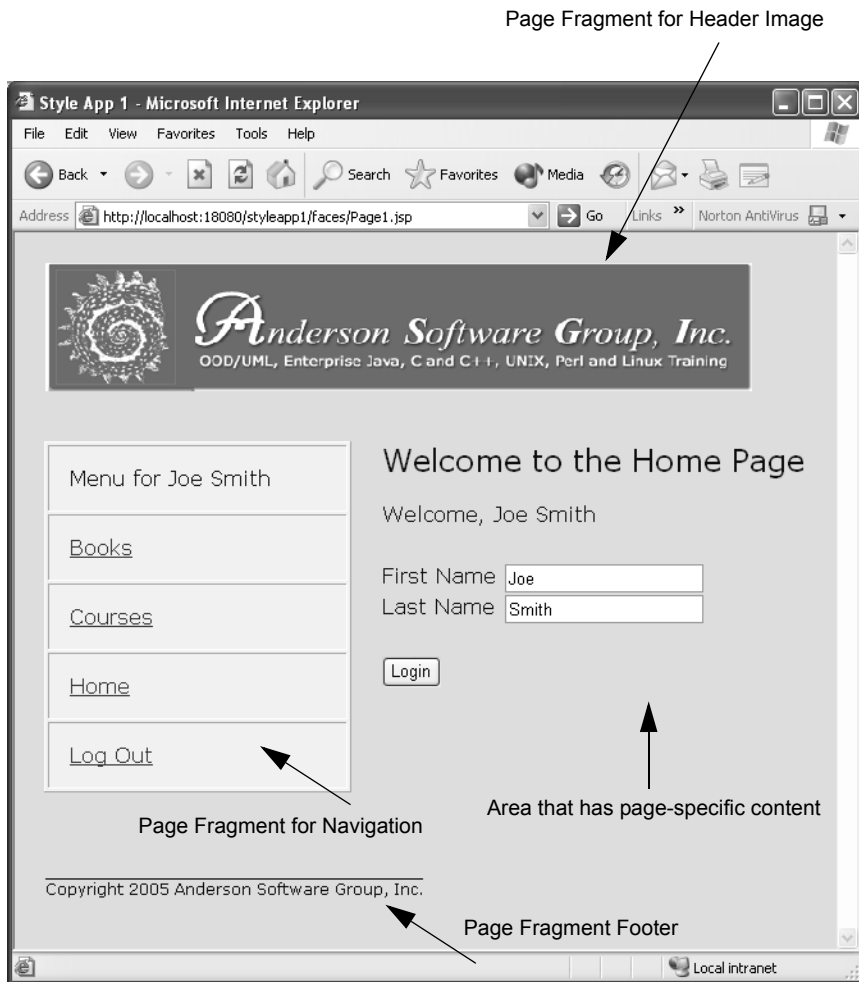


Figure B-5 Page layout using page fragments

The Books and Courses pages display data table components. The data is stored in application scope and loaded when the user accesses the page. As in the previous examples, the projects use a style sheet to control some basic style properties. Let's begin!

Create a New Project

1. In Creator's Welcome Page, click Create New Project. Specify the project name as **StyleApp1**. The selected project type is J2EE Web Application. Click OK.
2. After creating the project, Creator comes up in the design view of the editor pane. Click anywhere in the design canvas. In the Properties window, change the `Title` attribute to **Style App 1**, followed by `<Enter>`.

Default Style Sheet

Now you'll import a style sheet to replace the default style sheet.

1. In the Project Navigator window, expand the Resources folder. You'll see the default style sheet, **stylesheet.css**.
2. Double click the **stylesheet.css** file name. Creator brings up this file in the editor pane.
3. Copy and paste the contents of file **FieldGuide/Examples/Beyond2/files/ASGStyleSheet.css**, replacing the default **stylesheet.css**. The new style sheet is shown in Listing B.1.

Listing B.1 stylesheet.css

```
/* custom style rules */
body {
    background-color: #dddddd;
    color: navy;
    font-family: Verdana, Helvetica, sans-serif;
}

.headingStyle {
    font-size: 150%;
}

.footerStyle {
    font-size: 75%;
    border-top: 1px solid #000;
}

.gridStyle {
    background-color: #ffff88;
}
```

Listing B.1 stylesheet.css (continued)

```
td, th {
    padding-left: 1em;
    padding-top: 1em;
    padding-bottom: 1em;
    padding-right: 1em;
}

/* Style rules to make data tables look better */

.list-header {
    background-color: #eeeeee;
    font-size: 110%;
    font-weight: normal;
}

.list-paging-header {
    text-align: center;
}

.list-paging-footer {
    text-align: center;
}

.list-row-even {
    font-size: 70%;
}

.list-row-odd {
    background-color: #eeeeee;
    font-size: 70%;
}

.firstColumn {
    width: 65%;
    text-align: left;
}

.secondColumn {
    width: 35%;
    text-align: center;
}

.tableStyle {
    width: 400px
}
```

Listing B.1 stylesheet.css (continued)

```
/* Style rules for message severity levels */

.infoMessage {
  color: black;
}

.warnMessage {
  color: orange;
  font-weight: bold;
}

.errorMessage {
  color: red;
  font-style: italic;
  font-size: 75%;
}

.fatalMessage {
  color: red;
  font-style: italic;
  font-weight: bold;
}
```

4. Save the changes by selecting File > Save All from the main menu.
5. Return to the design canvas by selecting the tab labeled **Page1.jsp** above the editing pane. You'll note that the background color of the design canvas reflects the rule changes for `body`.

Let's briefly examine some of these style rules.

body: By setting property `color`, `background-color`, and `font-family`, you change these properties for all pages that you create in your project. By specifying three comma-separated values for `font-family`, browsers will use successive values if the primary `font-family` value is not available.

footerStyle: You apply this rule to text that appears on the bottom of each page. It's smaller and has a 1-pixel line above it. It's used to display a copyright notice.

td, th: these are HTML elements (table data and table heading) embedded in HTML `<table>` elements. JSF standard components grid panel and data table render with HTML `<table>` elements. The padding properties add space around table cells, improving appearance.

gridStyle: You apply this rule to the grid panel component used to hold navigation links. This style rules changes a component's background color (to a muted yellow).

.list (various): You'll note that the style rules for data tables are slightly modified from the default style sheet. The `.list-header` rule makes the `font-size` property larger (110%) and the `font-weight` normal. The text in the rows (style rules `.list-row-even` and `.list-row-odd`) is smaller (70%). Rule `.firstColumn` is fixed at 65% for width and `.secondColumn` is fixed at 35%. Finally, rule `.tableStyle` sets the `width` property to 400 pixels (400px).

By default, Creator sets the data table component's attribute `headerClass` to `list-header`. It sets the data table's attribute `rowClasses` to `list-row-even` and `list-row-odd`. The JSF data table component cycles through the style rules for attribute `rowClasses`. This achieves the striped-effect for the data table's rows (since `.list-row-odd` and `.list-row-even` specify different background colors).

Furthermore, you can control the look of a data table's columns using the same approach. Here we've specified style rule `.firstColumn` with a width of 65% and `text-align` left. Style rule `.secondColumn` has width 35% and `text-align` center. When you set the data table's attribute `columnClasses` to `firstColumn` and `secondColumn`, JSF cycles through these style rules for its columns. Then, when you also apply style rule `.tableStyle` to a data table's `styleClass` attribute, you fix the width to 400 pixels. This allows the cell's contents to remain a standard width.



Creator Tip

You can see the application of these style rules in example project `StyleApp1` from the book's download.

.errorMessage: Creator applies this rule to all message components for error messages, making the text italic and smaller (75%).

Add SessionBean1 Properties

Throughout this web application, you will keep track of the user's login status as well as the user's first name and last name. You store all of this data as properties of `SessionBean1`, since each session maintains its own login data. Furthermore, you want to access this data from any page. The next step, then, is to add three properties to managed bean `SessionBean1`, as follows.

1. In the Project Navigator window, expand `Java Sources > styleapp1`.
2. Right-click on `SessionBean1.java` and select `Add > Property`. Creator displays the `New Property Pattern` dialog.
3. Fill in the fields as follows. Specify `firstname` for Name. Select `String` in the dropdown list for Type. Select `Read/Write` for Mode. Make sure the options

Generate Field, Generate Return Statement, and Generate Set Statement are all checked.

4. Click OK.
5. Repeat this process and add two more properties. Add property **lastname** (specify type **String**) and **loggedIn** (specify type **Boolean**).

Now you need to provide initial values for these three properties in the **SessionBean1.java** constructor.

1. From the Project Navigator window, double-click file **SessionBean1.java**. This brings it up in the Java source editor.
2. Copy and paste the code from the download file **FieldGuide/Examples/Beyond2/snippets/StyleApp1_SessionBean1_constructor.txt**. Place it after the comment, as follows.

```
// Additional user provided initialization code  
firstname = "";  
lastname = "";  
loggedIn = new Boolean(false);
```

Create Page Fragments

You'll now create three page fragments that you will use to construct a uniform page layout for your project.

1. Make sure that **Page1.jsp** is displayed in the design canvas.
2. From the JSF Standard Components palette, select Page Fragment Box and drop the component onto the design canvas. Don't worry about position right now.
3. Creator pops up the Select Page Fragment dialog. Since you do not have any page fragments defined, the Page Fragment selector component is empty. Click the button Create new page fragment, as shown in Figure B-6.
4. Creator then pops up the Create Page Fragment dialog. Supply the page fragment name **ASGHeader** and click OK.
5. Creator returns to the Select Page Fragment dialog. Fragment ASGHeader appears in the Page Fragment selector component. Click OK.

Creator Tip

Don't use the name "header." A runtime exception occurs if a page fragment is named exactly "header."



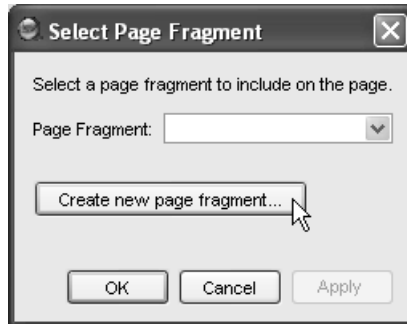


Figure B–6 Select Page Fragment dialog

6. Select the page fragment and resize it so that is approximately 30 grids wide by 7 grids high. (You'll be resizing it again later, but you want it big enough to hold the image that you'll add to it.)

Let's look at what Creator has done so far. In the Project Navigator view under Web Pages, you'll note a new element entitled **ASGHeader.jspf**, the page fragment you just created. Under the Java Sources > styleapp1 folder there's also a page fragment bean, **ASGheader.java**. Now look at the Application Outline view for **Page1.jsp**. You'll see an HTML `<div>` element and an embedded `directive.include` element (the page fragment). On the design canvas for **Page1.jsp**, you'll see a page fragment box with the text "Included Content Here." There's nothing in the page fragment yet. Let's configure it now.

1. From the Project Navigator window, double-click the file **ASGHeader.jspf**. Creator displays the page fragment in the design canvas.
2. Click anywhere on the text Included Context Here, right-click, and select Delete from the context menu. This removes the placeholder text Creator generated for you.
3. From the JSF Standard Components palette, select Image. Drop it onto the design canvas in the top left-most part of the canvas. Creator pops up the Select File or URL dialog.
4. Make sure the File tab is selected.
5. Browse to the download directory and select file **FieldGuide/Examples/Beyond2/images/anderson_software_group.gif**. Click OK. Creator copies the image file to the Resources folder of your project and places the image on the design canvas.
6. Make sure that the image is selected. In the Properties window, change attribute `id` to **asgHeaderImage1**.

Creator Tip

When using page fragments, you should make sure that the id attributes of its components do not conflict with the id attributes of the components on the current page. The best way to do this is to change the id attributes of page fragment components to meaningful names that include the page fragment's name, as shown above.



7. Resize the page fragment so that it includes the image. Save the file by selecting File > Save All from the main menu.
8. Check the position of the image by examining the source for this page fragment. (Select the tab labeled Source at the bottom of the editor pane.) The position values for `top` and `left` should both be 0, as follows.

```
<h:graphicImage binding="#{ASGHeader.asgHeaderImage1}"
  height="100" id="asgHeaderImage1"
  style="left: 0px; top: 0px; position: absolute"
  url="resources/anderson_software_group.gif"
  value="resources/anderson_software_group.gif" width="552"/>
```

Let's return to **Page1.jsp** and reposition the page fragment.

1. Select the tab labeled **Page1.jsp** from the top of the editor pane.
2. Position the page fragment image on the page so it is in the top left-most part of the canvas. The editor pane should look something like Figure B-7.

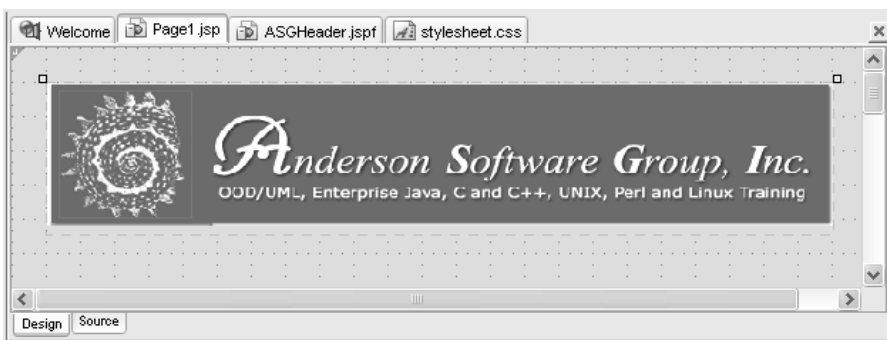


Figure B-7 Design view of Page1

You've created the first page fragment (the page's header) and placed it on the page. You'll now create a second page fragment to hold the web application's navigation links.

1. From the JSF Standard Components palette, select Page Fragment Box and drop the component onto the design canvas.
2. Creator pops up the Select Page Fragment dialog. Click the button Create new page fragment.
3. Creator then pops up the Create Page Fragment dialog. Supply the page fragment name **ASGLeftMenu** and click OK.
4. Creator returns to the Select Page Fragment dialog. Fragment ASGLeftMenu appears in the Page Fragment selector component. Click OK.
5. Select the page fragment and resize it so that is approximately 20 grids wide by 15 grids high.

The left menu page fragment consists of a grid panel component (so we can make the menu's background color different) which in turn contains an output text component for a heading and a set of link action components for event handling and navigation.



Creator Tip

It's important to add the grid panel components in the correct order, since (as of this writing) Creator does not provide easy visual rearrangement of components inside a grid panel.

Let's add the components to the ASGLeftMenu page fragment now.

1. From the Project Navigator window, double-click the file **ASGLeftMenu.jspf**. Creator displays the page fragment in the design canvas.
2. Click anywhere on the text Included Context Here, right-click, and select Delete from the context menu.
3. From the JSF Standard Components palette, select Grid Panel. Drop it onto the design canvas in the top left-most part of the canvas.
4. Make sure the grid panel is selected. In the Properties window, change the `id` attribute to **asgLeftGridPanel1**.
5. Under Appearance (in the Properties window), set attribute `styleClass` to **gridStyle**. (Rule `.gridStyle` is in the custom style sheet added earlier.)
6. Set attribute `border` to **1**.

The grid panel will hold an output text component and four link action components. Add them one at a time by dropping them directly on the grid panel component in the Application Outline view. Once these components are added, you can configure them.

1. From the JSF Standard Components palette, select Output Text and drop it onto the grid panel component in the Application Outline view.
2. Return to the JSF Standard Components palette and select Link Action. Drop it onto the grid panel component. The Application Outline view will show the second component you just added.
3. Repeat this and add three more link action components. The Application Outline view for the ASGLeftMenu page fragment should look like Figure B–8.

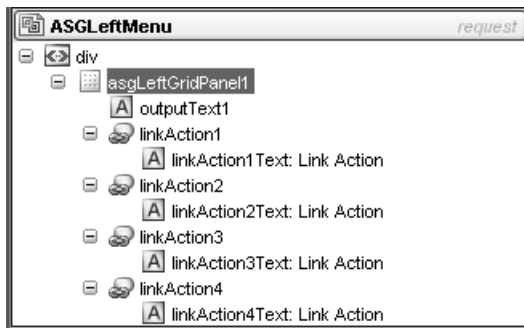


Figure B–8 Application Outline view for page fragment ASGLeftMenu

Now you'll configure each component, starting with the output text. (If the grid panel does not render correctly on the design canvas, switch views to **Page1.jsp**; then return to **ASGLeftMenus.jspf**.)

1. Make sure that **ASGLeftMenu.jspf** shows in the design canvas and select the output text component, `outputText1`.
2. In the Properties window, set attribute `id` to **asgLeftMenuHeader1**.
3. In the Properties window, click the small editing box opposite attribute value. Creator pops up an editing dialog. Specify the following value binding expression and click OK.

```
Menu for #{SessionBean1.firstname} #{SessionBean1.lastname}
```

Properties `firstname` and `lastname` were added earlier to **Session-Bean1.java**.

Now you'll configure each of the link action components (see Table B.1).

1. In the Application Outline view, select the first link action component, `linkAction1`.

2. In the Properties window, set attribute `id` to **booksAction**.
3. Now select the embedded output text component and in the Properties window, set attribute `id` to **booksActionText**.
4. In the Properties window, set attribute `value` to **Books**.

Follow the above procedure for each of the remaining link action components and their embedded output text components. Use the settings in the following table.

Table B.1 Link Action Components

<i>Component</i>	<i>Attribute</i>	<i>Setting</i>
<code>linkAction1</code>	<code>id</code>	<code>booksAction</code>
<code>linkAction1Text</code>	<code>id</code>	<code>booksActionText</code>
	<code>value</code>	<code>Books</code>
<code>linkAction2</code>	<code>id</code>	<code>coursesAction</code>
<code>linkAction2Text</code>	<code>id</code>	<code>coursesActionText</code>
	<code>value</code>	<code>Courses</code>
<code>linkAction3</code>	<code>id</code>	<code>homeAction</code>
<code>linkAction3Text</code>	<code>id</code>	<code>homeActionText</code>
	<code>value</code>	<code>Home</code>
<code>linkAction4</code>	<code>id</code>	<code>logoutAction</code>
<code>linkAction4Text</code>	<code>id</code>	<code>logoutActionText</code>
	<code>value</code>	<code>Log Out</code>

Figure B-9 shows the design view for **ASGLeftMenu.jspf** after making the above modifications.

Each of the link action components corresponds to a navigation case label. Configure each one as follows.

1. Select the link action component in the design canvas (issue successive clicks until the link action component is selected), right-click, and select **Edit Event Handler > action** from the context menu.
2. Creator displays file **ASGLeftMenu.java** in the source editor and puts the cursor at the newly generated event handler for the link action component.
3. Change each `return null` statement to the appropriate navigation label (refer to the code listing that follows).
4. Return to the design canvas for **ASGLeftMenu.jspf** and repeat Steps 1 through 3 for each link action component.

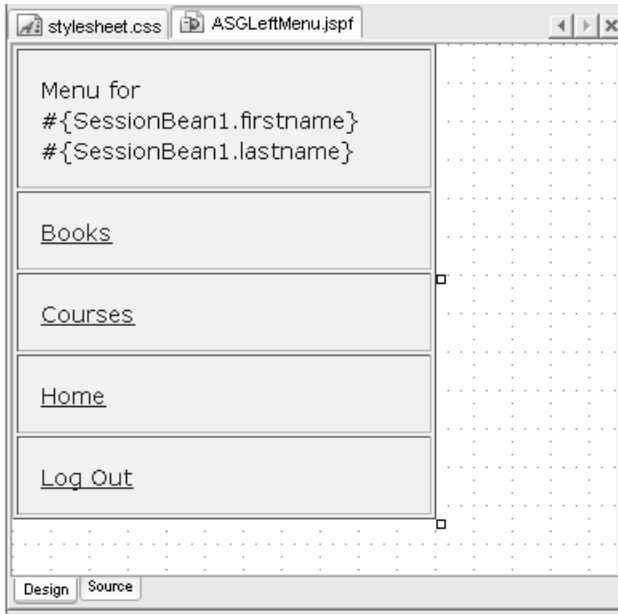


Figure B-9 Design view for page fragment ASGLeftMenu.jspf

5. For component `logoutAction`, copy and paste the event handler code from file **FieldGuide/Examples/Beyond2/snippet/StyleApp1_logoutAction.txt**. The added code for this event handler is bold.

```
public String booksAction_action() {  
    // User event code here...  
    return "Books";  
}  
  
public String coursesAction_action() {  
    // User event code here...  
    return "Courses";  
}  
  
public String homeAction_action() {  
    // User event code here...  
    return "Home";  
}
```

```

public String logoutAction_action() {
    // User event code here...
    getSessionBean1().setFirstname("");
    getSessionBean1().setLastname("");
    getSessionBean1().setLoggedIn(new Boolean(false));
    return "Home";
}

```

6. When you've finished configuring the link action components, return to the design canvas for **Page1.jsp** and adjust the page fragment to fit the newly configured grid panel.

It's time to create the third page fragment, which will hold a copyright notice.

1. Make sure that **Page1.jsp** is in the design canvas.
2. From the JSF Standard Components palette, select Page Fragment Box and drop the component onto the design canvas.
3. Creator pops up the Select Page Fragment dialog. Click the button Create new page fragment.
4. Creator then pops up the Create Page Fragment dialog. Supply the page fragment name **ASGFooter** and click OK.
5. Creator returns to the Select Page Fragment dialog. Fragment ASGFooter appears in the Page Fragment selector component. Click OK.
6. Select the page fragment and resize it so that is approximately 20 grids wide by 4 grids high.

Let's configure the ASGFooter page fragment now.

1. From the Project Navigator window, double-click the file **ASGFooter.jspf**. Creator displays the page fragment in the design canvas.
2. Click anywhere on the text Included Context Here, right-click, and select Delete from the context menu.
3. From the JSF Standard Components palette, select Output Text. Drop it onto the design canvas in the top left-most part of the canvas.
4. Make sure the output text is selected. Type in the text **Copyright 2005 Anderson Software Group, Inc.** (hey, we're forward-looking) followed by **<Enter>**.
5. In the Properties window, change attribute **id** to **footerText**.
6. Under Appearance (in the Properties window), set attribute **styleClass** to **footerStyle**. (Rule **.footerStyle** is in the custom style sheet added earlier.)
7. Save your project by selecting File > Save All in the main menu.
8. Return to the design canvas for **Page1.jsp** and position the footer page fragment near the bottom of the page at the left edge.

Deploy and Run

Deploy and run the project to check the placement of the page fragments on the page. Adjust them as necessary. Note that the menu title has no login information and that selecting the links simply resubmits the current page (you have not yet defined any navigation rules or even new pages!).

Add Output Text Components to Page1

The StyleApp1 web application's first page contains the three page fragments you created. Now you'll add text (using output text components), input fields, component labels, inline message components, and finally, a "Login" button. Figure B-10 shows the design canvas with these added components. Following best practices, you'll rename the components (changing their `id` attributes) so that they'll have meaningful names.

1. Make sure that **Page1.jsp** is in the design canvas.
2. From the JSF Standard Components palette, select Output Text and drop it onto the design canvas. Position it below the header image to the right of the navigation grid.
3. While it's still selected, type in the text **Welcome to the Home Page** followed by **<Enter>** to set its `value` attribute.
4. From the Properties window, change the component's `id` attribute to **homePageTitle**.
5. In the Properties window under Appearance, set attribute `styleClass` to **headingStyle**.
6. Now add a second output text component to the page. Position it directly below the `homePageTitle` component you just added.
7. In the Properties window, change its `id` attribute to **instructText**.

Note that you haven't set the `value` attribute; you'll set it programmatically in the page's constructor as well as the life cycle method `beforeRenderResponse()`, as follows.

1. Right-click inside the design canvas and select View Page1 Java Class. Creator brings up **Page1.java** in the source editor.
2. Copy and paste initialization code in the Page1 constructor to set the `instructText`'s `value` attribute. Its value depends on whether or not the user is logged in. Use file **FieldGuide/Examples/Beyond2/snippets/StyleApp1_Page1_constructor.txt**, as shown. The added code is bold.

```
// Additional user provided initialization code
// see if user is logged in
if (getSessionBean1().getLoggedIn().booleanValue()) {
```

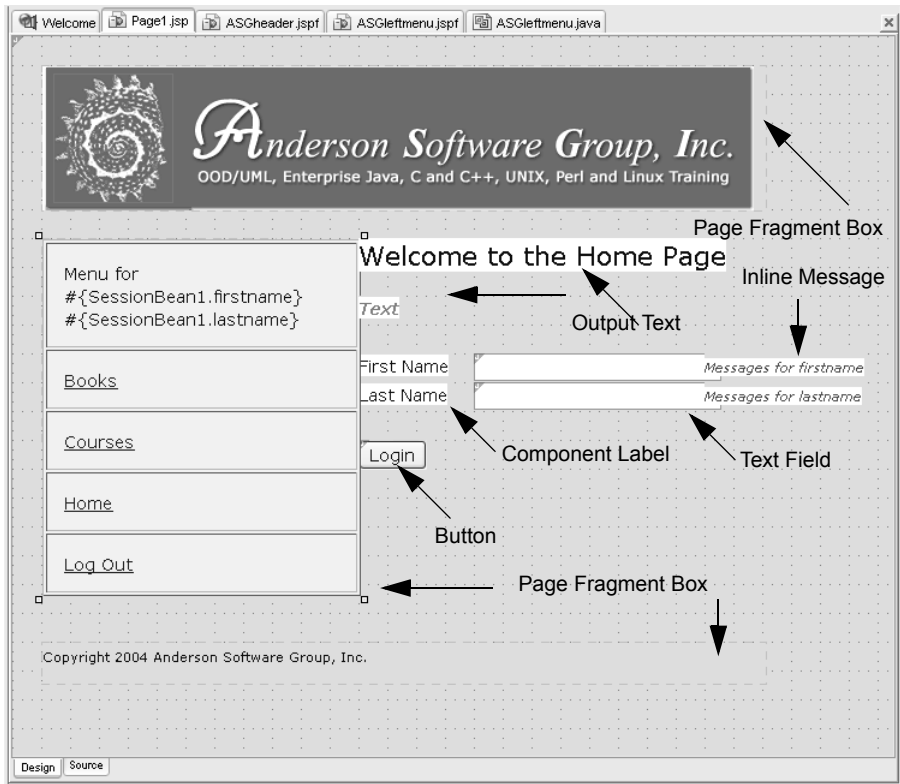


Figure B-10 Design view of Page1 showing the added components

```

instructText.setValue("Welcome, " +
    getSessionBean1().getFirstname() + " " +
    getSessionBean1().getLastname());
}
else instructText.setValue("Please Login");

```

- Copy and paste the same initialization code for method `beforeRenderResponse()`. (For more information on JSF life cycle methods, see "Life Cycle Issues" on page 183 of the Creator Field Guide book.) Place the method after

the constructor code in `Page1.java`. Use file `FieldGuide/Examples/Beyond2/snippets/SytleApp1_Page1_before.txt`, as shown.

```
protected void beforeRenderResponse() {
    // see if user is logged in
    if (getSessionBean1().getLoggedIn().booleanValue()) {
        instructText.setValue("Welcome, " +
            getSessionBean1().getFirstname() + " " +
            getSessionBean1().getLastname());
    }
    else instructText.setValue("Please Login");
}
```

Add Components for Input

The login process simply asks for a first name and last name from the user. These fields are required, but no validation of the names is performed. Each input field has a component label, an input text component, and an inline message component. You'll use Creator's "component linking" feature to link the component label and inline message to their respective input text component.

1. Return to the design canvas for Page1 by selecting the tab labeled `Page1.jsp` at the top of the editor pane.
2. From the JSF Standard Components palette, select Component Label and drop it onto the design canvas below component `instructText`.
3. Make sure the embedded output text component is selected and set its value to **First Name**.
4. Select component Text Field and drop it next the component label you just added.
5. In the Properties window, change its `id` attribute to **firstname** and check attribute `required`.
6. Select the text field component in the design canvas, right-click, and select Property Bindings. Creator pops up a Property Bindings dialog.
7. Under Select bindable property, make sure **value Object** is selected.
8. Under Select binding target, select **SessionBean1 > firstname String**. Click Apply and Close.
9. From the JSF Standard Components palette, select Inline Message and drop it next to the text field component you just added.

Both the component label and inline message components have an attribute `for`, which allows you to *link* the component. You can set the `for` attribute in the Properties window and supply the `id` attribute of the component to which you wish to create a link. More conveniently, Creator lets you link components

by selecting the component that has the `for` attribute, typing **<Ctrl-Shift>**, and dragging the cursor to the target component. Let's do this now.

1. Select the inline message component, type **<Ctrl-Shift>**, and drag the cursor to component `firstname` (the text field component). The design canvas displays *"Messages for firstname."*
2. Now select the component label, type **<Ctrl-Shift>**, and drag the cursor to component `firstname`.

Repeat the above sequence (steps 1-9 followed by steps 1 and 2) to add a second set of components for the last name. Add a component label (set the label's text to **Last Name**), a text field (set its `id` to **lastname**), and an inline message component. Link the component label and the inline message component to the text field. Be sure to bind the `value` attribute of component `lastname` to `SessionBean1` property `lastname`.

Add a Button Component

A button component allows the user to "officially login."

1. From the JSF Standard Components palette, select **Button** and drop it onto the design canvas below the text field for last name.
2. Set its `value` attribute to **Login**.
3. In the Properties window, change its `id` attribute to **loginButton**.
4. Double-click the component on the design canvas. Creator generates the event handler `loginButton_action()` and brings up **Page1.java** in the source editor.
5. Add the following event handler code. (Note: the property bindings you specified between the text field components and the session properties `firstname` and `lastname` mean that you don't need code in the event handler to set these properties.) The added code is bold.

```
public String loginButton_action() {
    // User event code here...
    getSessionBean1().setLoggedIn(new Boolean(true));
    return null;
}
```

Deploy and Run

We recommend that you deploy and run the application. Now you are prevented from selecting any of the navigation links as well as the login button until you provide information for the first and last name fields. Once you login, you can select the navigation links. The Log Out choice clears the fields and

makes it necessary to login again before selecting other links. As before, none of the links actually performs navigation.

Add Pages

Before you can specify navigation, you must create new pages. The easiest way to create new pages is using the Page Navigation's visual editor.

1. Bring up **Page1.jsp** in the design canvas.
2. Right-click in the background and select Page Navigation from the context menu. Creator brings up the Page Navigation editor.
3. Place the cursor anywhere in the navigation editor pane (in the background area) and right-click. From the context menu select New Page. Provide the name **BooksPage**. This creates a new page called **BooksPage.jsp**.
4. Repeat this process to create another page called **CoursesPage**.

You just created two new blank pages. You'll now add the three page fragments to each page, as well as an output text component to hold the heading.

1. Bring up each page in the design editor. (In the Project Navigator view, double-click the target web page.)
2. From the JSF Standard Components palette, select Page Fragment Box. Creator pops up a Page Fragment Selection dialog. Choose the desired page fragment from the dropdown menu and click OK.
3. Repeat this step for each of the page fragments. When you're finished, both pages will contain all three page fragment boxes.
4. Position the page fragments so that they are in the same position as the page fragments on **Page1.jsp**. You will also have to resize the fragment so that the components are properly displayed.
5. After you've added the page fragments, add an output text to each page (with attribute `styleClass` set to **headingStyle**). For **BooksPage.jsp**, set the output text's `value` attribute to **Books**. For **CoursesPage.jsp**, set the heading output text's `value` attribute to **Courses**. Be sure to change the default `id` attribute to a meaningful name.

Specify Page Navigation

The page fragment, **ASGLeftMenu.jspf**, contains the link action components for navigation. Since this page fragment is on *each page*, you must specify navigation rules for each page. In our example, this can get quite messy using the Page Navigation visual editor. Therefore, you're going to cheat! Basically, you want three navigation cases (remember, both the link action event handlers for Home and Log Out return navigation label **Home**):

1. Navigation label **Books** should bring up page **BooksPage.jsp**.
2. Navigation label **Courses** should bring up page **CoursesPage.jsp**.
3. Navigation label **Home** should bring up page **Page1.jsp**.

As it turns out, JSF provides an advanced navigation algorithm that allows wildcard expressions. You're going to define these three rules and then modify the navigation configuration in the editor and provide the wildcard expression.

Alternatively, you can completely specify the necessary rules using the visual editor. But, instead of having just three cases, you would need six cases (two cases for each of the three pages). Plus, you end up with a very spaghetti-like navigation diagram and spaghetti is never good when applied to programming projects.

1. Return to the Page Navigation editor by selecting the tab labeled **Page Navigation** from the top of the editor pane.
2. Select **Page1.jsp**, and drag the cursor to **BooksPage.jsp**, letting go when the cursor is inside the page.
3. Creator creates a navigation case label. Change the default name to **Books**.
4. Now create a second navigation case by selecting **Page1.jsp** and dragging the cursor to **CoursesPage.jsp**.
5. Change the default name to **Courses**.
6. Finally, select **BooksPage.jsp** and drag the cursor to **Page1.jsp**.
7. Change the default name to **Home**.

Here's the configuration file Creator generates for you.

```
<faces-config>
  <navigation-rule>
    <from-view-id>/Page1.jsp</from-view-id>
    <navigation-case>
      <from-outcome>Books</from-outcome>
      <to-view-id>/BooksPage.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>Courses</from-outcome>
      <to-view-id>/CoursesPage.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
```

```
<navigation-rule>
  <from-view-id>/BooksPage.jsp</from-view-id>
  <navigation-case>
    <from-outcome>Home</from-outcome>
    <to-view-id>/Page1.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
</faces-config>
```

Note that Creator generated two rules. One rule has two “cases,” and the other rules has a single case.

1. Modify the configuration file so that you have only one navigation rule with three navigation cases. Change the `<from-view-id>` element to `/*` (which matches any page).
2. Then add the three cases unchanged. Remove the unneeded `<navigation-rule>` element. Here is the modified file. (You can copy and paste from `FieldGuide/Examples/Beyond2/snippets/StyleApp1_navigation.txt`.)

```
<faces-config>
  <navigation-rule>
    <from-view-id>/*</from-view-id>
    <navigation-case>
      <from-outcome>Books</from-outcome>
      <to-view-id>/BooksPage.jsp</to-view-id>
    </navigation-case>

    <navigation-case>
      <from-outcome>Courses</from-outcome>
      <to-view-id>/CoursesPage.jsp</to-view-id>
    </navigation-case>

    <navigation-case>
      <from-outcome>Home</from-outcome>
      <to-view-id>/Page1.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

Deploy and Run

You’re finally ready to run the application with full navigation configured. Run the web application and test all of the navigation cases from different pages.

Session Timeout

There is still one more scenario to take care of in this web application. Suppose the session times out and the current page displayed in **Page1.jsp**? When a session times out and the application server receives a new request, it basically builds a new session for the current request. Thus, it newly instantiates the `SessionBean1` object (which resets property `loggedIn` to false). Assuming that the user clicked on one of the navigation links or the button, the validation for the text fields fails (remember, `required` is checked for these). The user is prevented from issuing any action until a new login sequence is completed. This is fine.

However, suppose the web application times out while either **BooksPage.jsp** or **CoursesPage.jsp** is displayed? In this case, the session is still newly instantiated, the user is no longer logged in, but no corrective action can take place until the user navigates to **Page1.jsp**. Instead, we'd like the user to immediately be taken to **Page1.jsp** so that he or she can login anew.

JSF provides a `redirect()` method that takes the user to a specific URL. How do you detect a session timeout? In our application, a session timeout is guaranteed to mark the user not logged in because the `SessionBean1` constructor contains code that initializes this property to false. Therefore, in the constructor of each page that is not the login page (that is, not **Page1.jsp**), you can see if the user is still logged in. If not, then you issue a `redirect()` to **Page1.jsp**.

1. In the Project Navigator view, open folder Java Sources > styleapp1. Double-click **CoursesPage.java** to bring it up in the source editor.
2. Add the following code to the constructor. (Add it after the comment, as shown). Copy and paste from **FieldGuide/Examples/Beyond2/snippets/PageFrag_timeout.txt**. The added code is bold.

```
// Additional user provided initialization code
// If the current session has timed out,
// then getLoggedIn() will return false.
// Redirect to the login page . . .
if (!getSessionBean1().getLoggedIn().booleanValue()) {
    try {
        getExternalContext().redirect("Page1.jsp");
    } catch (Exception e) {
        throw new FacesException(e);
    }
}
```

3. Repeat steps 1 and 2 for file **BooksPage.java**.

To test the timeout detection code, it's handy to change the default session timeout value. Here's how to change it to one minute (the unit for this parameter is minutes).

1. In the Project Navigator view, right-click the project name and select Show FileSystem View.
2. Expand nodes `src > web > WEB-INF`.
3. Double-click file **web.xml**. Creator brings it up in the source editor.
4. Add the following configuration information *in front of* the `Welcome File List` information. Copy and paste from **FieldGuide/Examples/Beyond2/snippets/StyleApp1_webxml_timeout.txt**.

```
<!-- Set global default timeout to 1 minute (for testing) -->
<session-config>
  <session-timeout>1</session-timeout>
</session-config>
```

Creator Tip

*Be sure to place the session configuration in the correct spot in **web.xml**. Otherwise, your project may not deploy.*



5. From the Project Navigation window, select the top-level project name (StyleApp1), right-click, and choose Clean Project. This insures that the newly-edited **web.xml** file is used the next time you build and run your project.
6. Deploy and run the application, testing the session timeout scenario.

Wait! There's More . . .

In the sample application that is included in the downloaded zip file, project StyleApp1 contains additional components. Included are JavaBeans objects instantiated in application scope to populate the data table components on pages **BooksPage.jsp** and **CoursesPage.jsp**. The technique used here is exactly the same as project DataTable2 (see "Data Table 2" on page 17 of *Beyond the Book*).

Noteworthy is that the data table components use style classes for attribute `columnClasses` to control the relative width of the columns and text alignment. Also, the data table's `styleClass` attribute specifies a constant width (400px). Thus, the fixed width, in conjunction with the `columnClasses` attribute, control the formatting of the table. JSF applies the style classes to the columns in order, cycling through the list if there are more columns than style

rules. Therefore, it applies `firstColumn` to column 1 and the second style class (`secondColumn`) to column 2.



Creator Tip

To conserve space, we “cleaned” project StyleApp1 before posting it in the examples. Therefore, the data tables which are bound to JavaBeans components do not render correctly (they appear red). To fix this, build the application. This creates the class file required by Creator to render the data tables. Close the project and reopen it; the data tables should now render correctly.

Here’s how to open and build the project.

1. Save and close the project if there’s any currently open in Creator.
2. From the Welcome window, select Open Existing Project and browse to **FieldGuide/Examples/Projects/StyleApp1** in your book’s download directory. Select Open.
3. From the main menu bar, select Build > Build project. This will create the necessary class file to render the data tables correctly in the design canvas.
4. Save the project (select File > Save All from the main menu) and then open it again in Creator.
5. From the Project Navigator window under Web Pages, double-click file **BooksPages.jsp**. Creator brings up this page in the design canvas, as shown in Figure B-11.

We added a data table component to this page. Take a moment to examine this in the IDE.

1. From the Application Outline view, select the data table component (make sure you select the top-level data table component, `dataTable1`).
2. Now look at the Properties window. You’ll see that attribute `rowClasses` is set to `list-row-even, list-row-odd`, which provides the striped-effect for the rows. (Creator provides this by default.)
3. Attribute `columnClasses` is set to `firstColumn, secondColumn`, which provides relative width values and text alignment settings.
4. Now scroll down the Properties window to attribute `value` (under Data). Click the small editing box opposite `value`. Creator pops up an editing box. Note that the `value` attribute is bound to

```
#{ApplicationBean1.bookData}
```

which is an array instantiated in application scope.

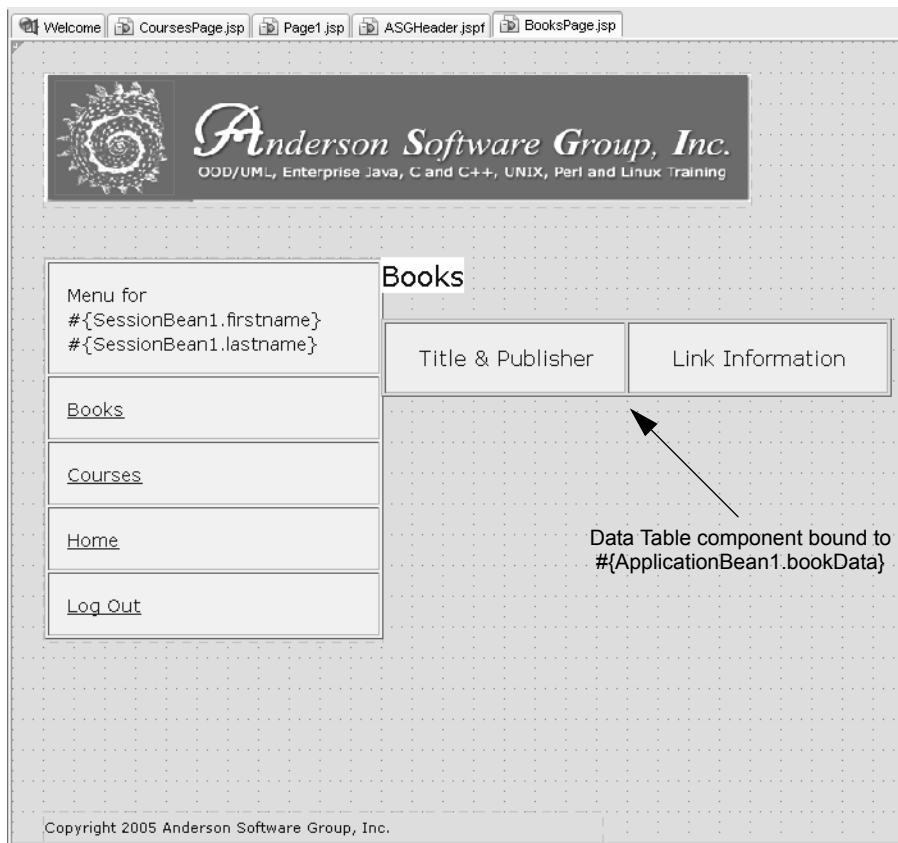


Figure B-11 Design view of BooksPage.jsp

Object `bookData` is an application scope array consisting of JavaBeans components. You can examine the source for **Book.java**, as follows.

1. From the Project Navigator window, open the Java Sources folder and the default package folder.
2. Double-click file **Book.java**. Creator displays this in the source editor. Here is its source (we removed the Creator-generated comments).

Listing B.2 Book.java

```
package styleappl;

public class Book {
    private String title;
    private String publisher;
    private String publishDate;
    private String linkInfo;
    private String linkDescription;

    public Book() {}

    public Book(String title, String publisher,
                String publishDate, String linkInfo,
                String linkDescription) {
        this.title = title;
        this.publisher = publisher;
        this.publishDate = publishDate;
        this.linkInfo = linkInfo;
        this.linkDescription = linkDescription;
    }

    public String getTitle() {
        return this.title; }

    public void setTitle(String title) {
        this.title = title; }

    public String getPublisher() {
        return this.publisher; }

    public void setPublisher(String publisher) {
        this.publisher = publisher; }
```

Listing B.2 Book.java (*continued*)

```
public String getPublishDate() {
    return this.publishDate; }

public void setPublishDate(String publishDate) {
    this.publishDate = publishDate; }

public String getLinkInfo() {
    return this.linkInfo; }

public void setLinkInfo(String linkInfo) {
    this.linkInfo = linkInfo; }

public String getLinkDescription() {
    return this.linkDescription; }

public void setLinkDescription(String linkDescription) {
    this.linkDescription = linkDescription; }

}
```

The `bookData` array is instantiated in **ApplicationBean1.java**. Here is a partial code listing of its instantiation.

```
// Additional user provided initialization code
bookData = new Book[] {
    new Book("Java Studio Creator Field Guide",
        "PrenticeHall (Sun Microsystems Press)", "June 2004",
        "http://www.asgteach.com/books/creator_field_guide.htm",
        "More Info"),
    new Book("Enterprise JavaBeans Component Architecture",
        "PrenticeHall (Sun Microsystems Press)", "March 2002",
        "http://www.asgteach.com/books/enterprise_java_beans.htm",
        "More Info"),
    . . .
};
```

This JavaBeans component has five properties that are bound to the various embedded components in the data table component on **BooksPage.jsp**. Recall that to bind a property to a data table component, you specify `#{current-Row.property_name}`. Let's look at one embedded component now.

1. From the Application Outline view, select component `outputText1` under `column1` of the data table component.

2. In its Properties window, you'll see that attribute `value` is set to

```
#{currentRow.title}
```

which binds to the `title` property of the current row of the `bookData` array.

You may want to examine several other of the embedded components under each column. Note that component `column1` contains four embedded output text components. One is used for the header text, one contains the HTML element `
` for formatting, and the other two are bound to properties in the `bookData` array.

Component `column2` also contains several components, including a hyperlink component.

The Courses page (**CoursesPage.jsp**) contains a similar data table component with a similarly defined JavaBeans component array instantiated in application scope. Its source is in file **Course.java** (under the Java Sources hierarchy) and the `courseData` array is instantiated in the `ApplicationBean1` constructor. The data table component on **CoursesPages.jsp** is bound to the `courseData` array with the expression

```
#{ApplicationBean1.courseData}
```

Again, the embedded components use `#{currentRow.property_name}` to bind to the Course JavaBeans component's properties.