



BEA WebLogic Portal™®

UI Design Guide

Version 9.1
UI Design Guide
December 2005

Copyright

Copyright © 1995-2006 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software is protected by copyright, and may be protected by patent laws. No copying or other use of this software is permitted unless you have entered into a license agreement with BEA authorizing such use. This document is protected by copyright and may not be copied photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form, in whole or in part, without prior consent, in writing, from BEA Systems, Inc.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE DOCUMENTATION IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA SYSTEMS DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE DOCUMENT IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks and Service Marks

Copyright © 1995-2006 BEA Systems, Inc. All Rights Reserved. BEA, BEA JRockit, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop, Built on BEA, Jolt, JoltBeans, SteelThread, Top End, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA AquaLogic, BEA AquaLogic Data Services Platform, BEA AquaLogic Enterprise Security, BEA AquaLogic Interaction, BEA AquaLogic Interaction Analytics, BEA AquaLogic Interaction Collaboration, BEA AquaLogic Interaction Content Services, BEA AquaLogic Interaction Data Services, BEA AquaLogic Interaction Integration Services, BEA AquaLogic Interaction Process, BEA AquaLogic Interaction Publisher, BEA AquaLogic Interaction Studio, BEA AquaLogic Service Bus, BEA AquaLogic Service Registry, BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Kodo, BEA Liquid Data for WebLogic, BEA Manager, BEA MessageQ, BEA Salt, BEA WebLogic Commerce Server, BEA WebLogic Communications Platform, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Enterprise Security, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Java Adapter for Mainframe, BEA WebLogic JDriver, BEA WebLogic Log Central, BEA WebLogic Mobility Server, BEA WebLogic Network Gatekeeper, BEA WebLogic Personalization Server, BEA WebLogic Personal Messaging API, BEA WebLogic Platform, BEA WebLogic Portlets for Groupware Integration, BEA WebLogic Real Time, BEA WebLogic RFID Compliance Express, BEA WebLogic RFID Edge Server, BEA WebLogic RFID Enterprise Server, BEA WebLogic Server Process Edition, BEA WebLogic SIP Server, BEA WebLogic WorkGroup Edition, BEA Workshop for WebLogic Platform, BEA Workshop JSP, BEA Workshop JSP Editor, BEA Workshop Struts, BEA Workshop Studio, Dev2Dev, Liquid Computing, and Think Liquid are trademarks of BEA Systems, Inc. Accelerated Knowledge Transfer, AKT, BEA Mission Critical Support, BEA Mission Critical Support Continuum, and BEA SOA Self Assessment are service marks of BEA Systems, Inc.

All other names and marks are property of their respective owners.

Contents

Introduction

UI Development Tools	1
UI Taxonomy and Control Tree Structure	1
HTML vs. XHTML	1
Skin/Skeleton Separation	1
Popular Reference Materials	1
How to Use this Document	2

Rendering Portals

Interaction with Content Types	1
How Look & Feel Determines Rendering	2
The Look & Feel File	4
The Portal File	7
Location of the Look & Feel Resources	8
Note About Portlet Titlebar Icons	9
Look & Feel Overrides	9
Classifications and Localizations	10
Overriding Look & Feel with Themes	10
Overriding Look & Feel with Properties	12
Summary	12
How Portal Components Are Rendered	13
Overview	13

Rendering Lifecycle of a Book	14
Selecting Look & Feel	14
Adding a Book to a Portal	15
JavaScript in Menus	29
CSS Styles in the Example	33
Changing Look & Feel	34

Creating a New Style

Working with Skins

Modifying the Title Bar	1
Modifying Fonts	1
Adding a Style	1
Additional Exercises	1
Changing Icons	1
Working with Skins: Best Practices	1

Modifying Skeletons

Adding Text to a Title Bar	1
Adding a Title Bar	1
Adding a Subtitle Bar	1
Additional Exercises	1
Adding a Subtitle Bar	1

Working with an Existing Look-and-Feel

Creating the Element	1
Adding the Element to the Look-and-Feel	1
Adding a Behavior	1
Overriding and Extending a Deployed Look-and-Feel	1

Additional Exercises	1
Integrating Portlet Content Styles with a Look and Feel	
Overview	1
Linking Dependencies to a Portlet.	1
Overriding Skins	1
Working with Themes, Classifications, and Localizations	
Overview	1
Configuring Themes, Classifications, and Localizations in a New Config File	1
Specializations	1
Matching Client Classes	1
Working with Genes	
Overview	1
What are Genes	1
How Genes Work	1
Exercise: Using Genes to Change a Color Set.	1
Integrating Third Party Multi-level Menus	
Overview	1
Impact on Control Tree Rendering	1
Exercise: Integrating Third-Party Multi-Level Menus	1
UI Design Resources	

Introduction

Brief description of what UI design is and what it entails

UI Development Tools

UI Taxonomy and Control Tree Structure

HTML vs. XHTML

Skin/Skeleton Separation

Different models for using these components together

Popular Reference Materials

Ideally, we'd like to identify a common reference with which most users would be familiar. This will reduce our need to explain standard concepts and terminology. If you can identify such a resource, please let me know

How to Use this Document

Rendering Portals

This document details how the portal framework turns a portal you develop in WebLogic Workshop (see Figure 1) into the portal desktop visitors see in a browser (see Figure 2). The goal of describing the portal framework is to help you develop and troubleshoot your portals. These topics enable you to look at a rendered portal in a browser and understand which pieces of the underlying framework you need to modify to get the results you want. In addition, the Look & Feel Editor is discussed. The Look & Feel Editor lets you interactively modify the text styles used by a portal.

The topics in this document describe key portal framework components and walk you through the portal rendering process. These topics include:

[How Look & Feel Determines Rendering](#)

Describes how look & feel determines how a portal desktop is rendered and what it looks like.

[How Portal Components Are Rendered](#)

Illustrates the rendering process, showing how a portal component is converted to HTML.

Portal file under development in WebLogic Workshop.

[image]

Portal rendered in HTML.

[image]

Interaction with Content Types

How Look & Feel Determines Rendering

When you build a portal in WebLogic Workshop, the look & feels you use are the key to how your portal is rendered and what it looks like when it is rendered. This topic shows you how the different pieces of the look & feel framework are combined and configured to provide what the portal framework needs to render the look & feel in HTML.

This topic contains the following sections:

Overview

The Look & Feel File

The Portal File

Location of the Look & Feel Resources

The skin.properties File

Look & Feel Overrides

Summary

Overview

The look & feel encompasses the following:

Skin - A skin is a group of Cascading Style Sheet (CSS) files, framework images (mainly for portlet title bar icons), and JavaScript functionality that is used in the portal desktop when it is rendered in HTML. A portal Web project can have multiple skins. When you select a look & feel for a desktop, a specific skin is used. Following are example skin elements, Image, CSS Style, and JavaScript Functions:

Image

CSS Style

```
.bea-portal-button-float
{
```

```

}
.bea-portal-button-float img
{
    vertical-align: top;
    margin: 1px;
    border-style: solid;
    border-width: 1px;
    border-color: #666699;
}

```

JavaScript Function

```

function initPortletFloatButtons()
{
    var links =
        document.getElementsByTagName("a");
    for (var i = 0; i < links.length; i++)
    {
        if (links[i].className &&
            links[i].className == "bea-portal-button-float")
        {
            initPortletFloatButton(links[i]);
        }
    }
}

```

Skeleton JSPs - A skeleton is a group of JSPs that are used to render each component of the portal desktop as HTML, from the desktop to books and pages to portlet title bars. The skeleton provides the physical boundaries of the portal components and provides references to the images, CSS classes, and JavaScript functions from the skin needed to render the portal. A portal Web project can have multiple skeletons. When you select a look & feel for a desktop, a specific skin and skeleton is used.

A look & feel is represented by an XML file (with a .laf extension). As shown in the following figure, the .laf (avitek.laf) file is located in the lookandfeel folder of a portal project. In addition, the .laf file name (for example, avitek) can be selected in the Desktop properties panel.

Portal rendered in HTML.

Developers building portals with WebLogic Workshop are not the only users who can select the look & feel used by a portal desktop. While developers create look & feels and select the default look & feel used by a portal, portal administrators and visitors may ultimately determine the desktop look & feel. The following figures show how portal administrators and users can change the look & feel used by the desktop.

After a portal administrator creates a desktop in the WebLogic Administration Portal, the administrator can change the desktop look & feel on the Desktop Properties page, as shown in the following figure.

The Desktop Properties page

If visitor tools are enabled for the desktop, visitors can click the "Customize My Portal" link and change the desktop look & feel, as shown in the following figure.

Customizing a portal

The following section shows the contents of a look & feel XML-based .laf file and describes how it is used as the basis of portal desktop rendering.

The Look & Feel File

Look & feel files point to the specific skin and skeleton to be used for the overall desktop look & feel.

Look & feel files are stored in the following location:

<portal_Web_project>/framework/markup/lookandfeel. Following is the avitek.laf provided by BEA. The key attributes are highlighted.

Listing 2-1

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

<netuix:markupDefinition
xmlns:netuix="http://www.bea.com/servers/netuix/xsd/controls/netuix/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.bea.com/servers/netuix/xsd/controls/netuix/
  1.0.0 markup-netuix-1_0_0.xsd">
  <netuix:locale language="en"/>
  <netuix:markup>
    <netuix:lookAndFeel
      definitionLabel="avitek" title="avitek"
      description="The avitek look and feel"
      skin="avitek" skinPath="/framework/skins/"
      skeleton="default" skeletonPath="/framework/skeletons/"
      markupType="LookAndFeel" markupName="avitek"/>
  </netuix:markup>
</netuix:markupDefinition>
```

The following table describes the look & feel file's key attributes.

Look & Feel File Attributes

AttributeDescription

definitionLabelRequired. The unique label used to identify the look & feel for setting entitlements. Each look & feel in the portal Web project must have a unique definitionLabel. For best practices, use the same name as the markupName.

titleRequired. The string used to display the name in the Look & Feel drop-down fields in WebLogic Workshop, the WebLogic Administration Portal, and on the visitor tools page.

descriptionOptional. Description of the look & feel. The description is used in the WebLogic Administration Portal; when you select a look & feel in the portal Library, the description appears on the Look & Feel Properties page.

skinOptional. The name of the directory containing the skin you want to use.

If you do not set this attribute, the /framework/skins/default skin is used.

skinPathOptional. The path, relative to the portal Web project, to the parent directory of the skin directory.

If you do not set this attribute, the /framework/skins/<skin_attribute_name> skin is used.

If no skin attribute is set, the /framework/skins/default skin is used.

skeletonOptional. The name of the directory containing the skeleton JSPs you want to use.

If you do not set this attribute, the framework uses the default.skeleton.id path in the skin.properties file of the skin used.

If you do not set this attribute and no default.skeleton.id path is set in skin.properties, the /framework/skeletons/default skeleton is used.

skeletonPathOptional. The path, relative to the portal Web project, to the parent directory of the skeleton directory.

If you do not set this attribute, the framework uses the default.skeleton.path in the skin.properties file of the skin is used.

If you do not set this attribute and no default.skeleton.path is set in skin.properties, the /framework/skeletons/<skeleton_attribute_name> skeleton is used.

If you do not set this attribute and no skeleton attribute is set, the /framework/skeletons/<default.skeleton.id> skeleton is used.

If you do not set this attribute and no skeleton attribute is set, and skin.properties contains no default.skeleton.id, the /framework/skeletons/default skeleton is used.

markupTypeRequired. The name of the type of component. Must always be "LookAndFeel".

markupNameRequired. The name for the look & feel. Each look & feel in the portal Web project must have a unique markupName. For best practices, use the same name as the definitionLabel.

When you select a Look & Feel in the WebLogic Workshop Property Editor for a selected desktop, the look & feel XML is automatically added to the underlying XML in the .portal file, as shown in the following section.

The Portal File

The following example portal file, created with the Portal Designer, shows the inserted look & feel XML (in bold) from the .laf file. The portal file is a template with which multiple desktops can be created in the WebLogic Administration Portal. When used as a template, the portal file determines the default look & feel of any desktop created from it.

Listing 2-2

```
<?xml version="1.0" encoding="UTF-8"?>
<portal:root xmlns:html="http://www.w3.org/1999/xhtml-netuix-modified/1.0.0"
  xmlns:netuix="http://www.bea.com/servers/netuix/xsd/controls/netuix/1.0.0"
  xmlns:portal="http://www.bea.com/servers/netuix/xsd/portal/support/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.bea.com/servers/netuix/xsd/portal/support/1.0.0
portal-support-1_0_0.xsd">
  <portal:directive.page contentType="text/html; charset=UTF-8"/>
  <netuix:desktop definitionLabel="defaultDesktopLabel" markupName="desktop"
    markupType="Desktop" title="New Portal Desktop">
    <netuix:lookAndFeel definitionLabel="avitek" description="The avitek look
and feel"
      markupName="avitek" markupType="LookAndFeel" skeleton="default"
      skeletonPath="/framework/skeletons/" skin="avitek"
```

```
        skinPath="/framework/skins/" title="avitek"/>
<netuix:shell description="A header with a link and footer is included in this shell."
    markupName="headerFooterVisitor" markupType="Shell"
    title="Header-Footer Visitor Shell">
<netuix:head/>
<netuix:body>
    <netuix:header>
        <netuix:jspContent contentUri="/portlets/header/header.jsp"/>
    </netuix:header>

    [XML for books, pages, and portlets...]

    <netuix:footer>
        <netuix:jspContent contentUri="/portlets/footer/footer.jsp"/>
    </netuix:footer>
```

The look & feel XML is inserted when the Look & Feel property is set for the selected desktop in the Property Editor. For example, in the following figure, the look & feel called avitek is selected.

Figure 2-1 Selecting a Look & Feel

When the .laf file is inserted into the .portal file, its job is finished in the rendering process and the .portal file is used to set look & feel.

Location of the Look & Feel Resources

The look & feel attributes in the portal file tell the portal which skin and skeleton to use to render the portal in HTML. The portal in the previous example will use the following skin and skeleton resources:

SkinSkeleton

The `/css`, `/images`, and `/js` directories contain the CSS files, framework images (mainly portlet titlebar icons), and JavaScript files that will be used in the skin. The `skin.properties` file (discussed in the next section) contains references to these resources, and at rendering time those resource references are inserted into the HTML `<head>` region. You can name your skin resource directories anything you like as long as you reference them correctly in `skin.properties` (or `skin_custom.properties`).

Skins can also contain subdirectories for sub-skins, or themes (discussed in Look & Feel Overrides). The skeleton is made up of JSPs that map to and convert each portal component to HTML. The XML elements for the portal components in the `.portal` file determine the order in which the skeleton JSPs are called and rendered as HTML.

This figure shows a clipped view of the skeleton contents. The subdirectories shown are skeleton themes and skeletons used for mobile devices. The JSPs in the `/default` directory make up the "default" skeleton.

Themes are discussed in Look & Feel Overrides.

Note About Portlet Titlebar Icons

The icons used in portlet title bars are stored in the skin's `/images` directory. The portal framework reads the portal Web project's `WEB-INF/netuix-config.xml` file to determine which of these graphics to use for the portlet's different states and modes (minimize, maximize, help, edit, and so on).

Look & Feel Overrides

You can override the skin elements and skeletons on individual portal components so that those components have a different look & feel than the other portal components. For example, you can override the look & feel of a portlet so that it looks different than the other portlets on a page.

Classifications and Localizations

Overriding Look & Feel with Themes

As part of each skin or skeleton, you can create sub-skins and sub-skeletons called "themes." Themes contain all or part of the resources contained in a skin or skeleton. For example, a skin theme can contain a /css subdirectory with a single CSS file, and a skeleton theme can contain a single JSP to render a portlet titlebar. Themes can be used on books, pages, and portlets.

Each theme requires a .theme file located in <portal_Web_project>/framework/markup/theme/. Following is a sample theme file:

```
<?xml version="1.0" encoding="UTF-8"?>
<netuix:markupDefinition
xmlns:netuix="http://www.bea.com/servers/netuix/xsd/controls/netuix/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.bea.com/servers/netuix/xsd/controls/netuix/1.0.0
markup-netuix-1_0_0.xsd">
  <netuix:locale language="en"/>
  <netuix:markup>
    <netuix:theme
      name="alert"
      title="Alert Theme" description="A simple alert theme."
      markupType="Theme" markupName="alert"/>
  </netuix:markup>
</netuix:markupDefinition>
```

The theme file contains two key attributes:

name - The name attribute value tells the portal framework the name of the theme directory to look in to apply theme resources to the book, page, or portlet.

title - The title attribute value is used to populate the Theme drop-down list where it appears in the book, page, and portlet properties in the Portal Designer and in the WebLogic Administration Portal.

The theme XML is inserted in the .portal around the XML for the book, page, or portlet to which the theme applies.

The following figures show where you set a theme in WebLogic Workshop and in the WebLogic Administration Portal.

WebLogic WorkshopWebLogic Administration Portal

Theme selection for a book, page, or portlet does not depend on the look & feel selected for a desktop. All themes are available for selection for all look & feels, whether or not the skins and skeletons for the look & feels contain the selected theme. If a skin or skeleton does not contain the selected theme, the theme is ignored. If both a skin and a skeleton theme exists for the selected look & feel, both are used.

The following figures show an example theme directory structure for the theme file, a skin theme, and a skeleton theme:

Theme FileSkin ThemeSkeleton Theme

If skin or skeleton resources are not explicitly contained in a theme, the parent skin or skeleton resources are used. For example, if a skeleton theme uses only a JSP to render a portlet titlebar, the parent skeleton JSPs are used to render the rest of the portlet.

For skeletons, the ability to use parent resources is dependent on a file in the skeleton theme directory called skeleton.properties, which contains a single entry:

jsp.search.path: ., ..

where ., .. is a relative path to the theme's own skeleton JSPs and to the parent skeleton's JSPs.

In the parent skin, the `skin.properties` must contain path information to its skin themes in the following format:

```
theme.alert.search.path: alert/images, images
```

The name of the theme directory is the second entry in the property. The path to the theme images is set (`alert/images`), along with the path to the parent skin's images directory (`images`) in case the theme images are an incomplete subset of the necessary images.

Overriding Look & Feel with Properties

For any selected component in the Portal Designer, you can override CSS properties and the skeleton JSP used to render the component. With the portal component selected in the Portal Designer, set the property overrides you want in the Property Editor under Presentation Properties, as shown in the following figure.

Property Editor

When the portal desktop is rendered as HTML, the skeleton JSP you selected is used to render the component, and the style overrides you entered are automatically inserted into the XML of the `.portal` file.

Summary

The look & feel selected for a portal desktop serves as the basis for how the desktop is rendered in HTML. The look & feel XML file (`.laf`) points to a specific skin and a specific skeleton on the file system to use for rendering.

Skins are made up of framework images (like portlet titlebar icons), CSS files, and script files, such as JavaScript. Skeletons are JSPs that convert XML-based portal components to HTML.

Once a look & feel is selected, its XML is inserted into the `.portal` XML file, which is the primary XML file used to control desktop rendering (`.portlet` XML files are used to render portlets). The look & feel settings point to the file-based skin and skeleton resources that are used to generate and used in the rendered HTML.

The skin used in a look & feel contains a `skin.properties` and an optional `skin_custom.properties` file that contains references to all images, CSS files, and script files that are used by the skin. The

entries in `skin.properties` and `skin_custom.properties` are converted to HTML `<head>` entries so that any framework images, CSS styles, and script functions used in the HTML are recognized.

You can override the look & feel for any book, page, or portlet by using themes; and using the Portal Designer and Portlet Designer Property Editor you can override CSS styles, attributes, and the skeleton JSP used to render desktops, books, pages, and portlet title bars and windows.

How Portal Components Are Rendered

With the look & feel and shell selected for a portal desktop, the rendering service has the basic information it needs to convert a .portal XML file into a final HTML file.

This topic shows the rendering lifecycle, step by step, for a single portal component. The same rendering principles apply for all other portal components.

This topic includes the following sections:

Overview

Single File vs. Streamed Rendering

Rendering Lifecycle of a Book

Summary

Overview

There are three basic stages in the portal rendering process—a process that ultimately results in a portal desktop being displayed in a browser:

1. **Building a portal in XML:** In the portal development process, you use the Portal and Portlet designers in WebLogic Workshop to build .portal and .portlet files. Both types are XML files. As you build portals and portlets in WebLogic Workshop, the XML elements and attributes are automatically built under the surface.

The previous topics, *How Look & Feel Determines Rendering* and *How the Shell Determines Header and Footer Content*, described part of the XML-building process: how the look & feel and shell XML files are added to the portal XML file to provide rendering instructions.

2. **Portal XML elements mapped to JSP skeleton files:** The portal framework maps specific XML elements to specific JSP skeleton files. They are called skeleton files because they are used to render the physical boundaries and structure—the skeleton—of their portal components. For example, a portlet titlebar in a portlet XML file uses an element called `<netuix:titlebar>`. The portal framework knows to use the `titlebar.jsp` skeleton file to render the portlet titlebar.

3. JSP skeleton files and skin.properties are rendered as HTML: Each skeleton JSP file performs its own processing, such as retrieving property values you set in the WebLogic Workshop Property Editor (and were automatically added to the portal XML file), and generates the appropriate HTML for the portal component. The skin.properties and optional skin_custom.properties files for the selected look & feel are converted to image path entries, CSS file entries, and script file entries in the HTML <head> area.

The following figure is a simplified illustration of the rendering process.

Portal rendering process

This topic will expand on these three stages using the rendering lifecycle of a single portal component as an example.

Before going into greater detail on the rendering process, it is important to understand the difference between viewing a portal in the development environment (WebLogic Workshop) and viewing it in the administration/end user environment (WebLogic Administration Portal/browser). The three-stage rendering process occurs in slightly different ways in the two different environments. The following section describes the basic principles of each.

Rendering Lifecycle of a Book

This section illustrates the rendering lifecycle of a book, which will help you understand the rendering lifecycle of other portal components, such as pages and portlets.

This section contains the following topics:

1. Building a portal in XML
2. Portal XML elements mapped to JSP skeleton files
3. JSP skeleton files and skin.properties are rendered as HTML

1. Building a portal in XML

This section describes steps that populate and configure the .portal XML file.

Selecting Look & Feel

When you select the look & feel for a desktop, the look & feel file determines which skin and skeleton is used to render all desktop components. In the following example, the "avitek" look &

feel has been selected, which uses the "avitek" skin and the "default" skeleton. The look & feel XML is added to the .portal XML file.

Listing 2-3

```
<netuix:markup>
  <netuix:lookAndFeel
    definitionLabel="avitek" title="avitek"
    description="The avitek look and feel"
    skin="avitek" skinPath="/framework/skins/"
    skeleton="default" skeletonPath="/framework/skeletons/"
    markupType="LookAndFeel" markupName="avitek"/>
</netuix:markup>
```

The skin and skeleton come into play later in the rendering process, when the desktop is viewed with a browser. Before that happens, the book that will be used to illustrate the rendering process will be added to the portal.

Adding a Book to a Portal

In this section a book is added to the desktop in the .portal file and configured. Books can also be added by portal administrators in the WebLogic Administration Portal, which adds the book directly to the database.

The following figure shows a book control being dragged onto the desktop.

Dragging a control to the Desktop

After the book is added to the desktop, the book title is changed from "New Book" to "My Book," and the navigation style is set to Multi Level Menu, as shown in the following figure.

Navigation controls the way a book's sub-books and pages are accessed. The single-level menu provides text links/tabs to sub-books and pages, and the multi-level menu provides a drop-down

menu to access sub-books and pages. (Books must be added to books rather than to pages inside books for drop-down navigation to work. So in the following example, for the multi-level menu to produce a drop-down menu, you would need to drag a new book control into Main Page Book, right next to Page 1, as shown in the following figure.)

Arranging portal components

After the Navigation style is set on My Book, the following is what the book looks like in XML in the .portal file. If you add a book in the WebLogic Administration Portal, the XML is added to the database. This XML is used as the basis for the rendering of the book.

```
<netuix:book defaultPage="newPage.1" definitionLabel="my_book_3"
  markupName="book" markupType="Book" title="My Book">
  <netuix:multiLevelMenu
    description="This menu can navigate across may nested books."
    markupName="multiLevelMenu" markupType="Menu" title="Multi Level Menu"/>
    <!-- in this example, the nested page content has been removed -->
  </netuix:book>
```

2. Portal XML elements mapped to JSP skeleton files

When the desktop is viewed in a browser, the portal framework reads the XML elements and uses the skeleton path to map the desktop's XML elements to skeleton JSPs. The following examples show which elements in the book XML are mapped to skeleton JSPs and which skeleton JSPs are used to render the elements.

book XML - The highlighted elements are mapped to skeleton JSPs.

```
<netuix:book defaultPage="newPage.1" definitionLabel="my_book_3"
  markupName="book" markupType="Book" title="My Book">
  <netuix:multiLevelMenu
    description="This menu can navigate across may nested books."
```



```

markupName="multiLevelMenu" markupType="Menu" title="Multi Level Menu"/>
    <!-- in this example, the nested page content has been removed -->
</netuix:book>

```

skeleton - Referenced in the look & feel

```

/framework/skeletons/default/
book.jsp
multilevelmenu.jsp
submenu.jsp (referenced in multilevelmenu.jsp)

```

Once rendering has been handed off to the JSPs, the JSPs perform the tasks necessary for conversion to HTML. Following are the `book.jsp`, `multilevelmenu.jsp`, and `submenu.jsp` used in this example. Comments are added to describe what the JSPs are doing.

`book.jsp`

The `book.jsp` serves as a high-level container for the book's menu and the book's child books and pages. Comments in the JSP code are highlighted in bold text.

```

<%@ page import="com.bea.netuix.servlets.controls.page.BookPresentationContext,
                com.bea.netuix.servlets.controls.page.MenuPresentationContext"
%>
<%@ page session="false"%>
<%@ taglib uri="render.tld" prefix="render" %>
<render:beginRender>

```

<%-- The content inside the <render:beginRender> tag is processed first and ultimately renders whatever is inside it. In most cases, the skeletons produce an opening <div> HTML tag with specific attributes such as CSS classes.

The following block determines where the book falls in the desktop hierarchy (whether it is the top-level book or a nested book). It also sets the base name of the CSS class to use (bea-portal-book) and appends different endings to the base class to apply a different CSS class for each book context. Only processing, not HTML rendering, occurs in this block.

--%>

<%

```
BookPresentationContext book =
    BookPresentationContext.getBookPresentationContext(request);
MenuPresentationContext menu = (MenuPresentationContext)
    book.getFirstChild("page:menu");
String bookClass = "bea-portal-book";
String useBookClass = bookClass;
if (book.isDesktopBook())
{
    bookClass += "-primary";
    useBookClass = bookClass;
}
else if (book.isLikePage())
```

```

{
    useBookClass += "-invisible";
}
String bookContentClass = bookClass + "-content";
%>
<%-- The next block begins the actual HTML rendering, beginning with
the comment "Begin Book" followed by an opening <div> HTML
tag. Notice the JSP tags used before the closing bracket of the
<div> tag. These populate the div tag with style attributes.
The methods retrieve any presentation property override values
you entered in the WebLogic Workshop Property Editor for the book.
For the "class" attribute, the default value is useBookClass,
which earlier is set to "bea-portal-book". (If through getting
the context the book was found to be the top-level book, the value
of useBookClass would be "bea-portal-book-primary".)
With no overrides, the useBookClass variable will produce the
following HTML, because the book is acting like a page:

```

```

<div
    class="bea-portal-book-invisible"
>

```

The style sheet class is provided by the skin, and the CSS file containing the class is referenced in the skin's skin.properties file and added to the HTML <head> region.

```

--%>

```

```
<%-- Begin Book --%>
<div
  <render:writeAttribute name="id" value="<%= book.getPresentationId() %>"/>
  <render:writeAttribute name="class" value="<%=
    book.getPresentationClass()%>" defaultValue="<%= useBookClass %>"/>
  <render:writeAttribute name="style" value="<%= book.getPresentationStyle() %>"/>
>
<%-- The following JSP tag gets the names of the pages and books it will
display in its navigation menu, and based on the navigation element
used in the portal XML file (in this case netuix:multiLevelMenu),
uses the corresponding menu JSP (multilevelmenu.jsp) to render the menu
in this position of the HTML.
--%>
```

```
  <render:renderChild presentationContext="<%= menu %>"/>
<%-- The following block provides a <div> HTML container for the
book's content area--the child books and pages. Again, it uses
a JSP tag to set the style sheet "class" attribute.
--%>
```

```
  <%-- Begin Book Content --%>
  <div
    <render:writeAttribute name="class" value="<%=
book.getContentPresentationClass()%>" defaultValue="<%= bookContentClass %>"/>
    <render:writeAttribute name="style" value="<%= book.getContentPresentationStyle()
%>"/>
  >
</render:beginRender>
```

```

<%-- The closing </render:beginRender> tag signals the portal framework
      to stop rendering the book. After the book's children and their
      children are rendered, the portal framework uses the following
      <render:endRender> tag to close the book's parent HTML tags.
--%>

```

```

<render:endRender>
    </div>
    <%-- End Book Content --%>
</div>
<%-- End Book --%>
</render:endRender>

```

Following is a description of the `multilevelmenu.jsp`, which is used by the book to render the navigation menu for the book's child books and pages.

`multilevelmenu.jsp`

The `multilevelmenu.jsp` is rendered inside the book container and provides the boundaries for multi-level menus on books. This JSP also uses `submenu.jsp` to perform the actual rendering of the menu links. Comments are highlighted in bold text.

```

<%@ page import="com.bea.netuix.servlets.controls.window.WindowPresentationContext,
               com.bea.netuix.servlets.controls.page.BookPresentationContext,
               com.bea.netuix.servlets.controls.page.MenuPresentationContext,
               java.util.List,
               java.util.Iterator,
               com.bea.netuix.servlets.controls.page.PagePresentationContext,
               com.bea.netuix.servlets.controls.window.WindowCapabilities" %>
<%@ page session="false"%>

```

```
<%@ taglib uri="render.tld" prefix="render" %>

<!-- The following block determines where the book falls in the desktop
      hierarchy (whether it is the top-level book or a nested book). It
      also sets the base name of the CSS class to use (bea-portal-book)
      and defines different menu style classes by appending different
      endings to the base class. Only processing, not HTML rendering,
      occurs in this block.

--%>

<%
    BookPresentationContext book =
        BookPresentationContext.getBookPresentationContext(request);
    MenuPresentationContext menu =
        MenuPresentationContext.getMenuPresentationContext(request);
    String bookClass = "bea-portal-book";
    if (book.isDesktopBook())
    {
        bookClass += "-primary";
    }
    final String menuClass = bookClass + "-menu";
    final String menuContainerClass = menuClass + "-container";
    final String menuItemClass = menuClass + "-item";
    final String menuItemActiveClass = menuItemClass + "-active";
    final String menuItemLinkClass = menuItemClass + "-link";
    final String menuHookClass = menuClass + "-hook";
    final String menuButtonsClass = menuItemClass + "-buttons";
    List menuChildren = menu.getChildren();
```

```
%>
```

```
<render:beginRender>
```

<%-- The content inside the <render:beginRender> tag is processed first and ultimately renders whatever is inside it, such as opening <div> HTML tags with specific attributes and tables.

The following block creates a table cell, sets CSS styles on the <td> tag (based on the members defined in the previous block).

```
--%>
```

```
<%-- Begin Multi Level Menu --%>
```

```
<div class="bea-portal-ie-table-buffer-div">
```

```
<table border="0" cellpadding="0" cellspacing="0" width="100%">
```

```
<tr>
```

```
<td class="<%= menuContainerClass %%" align="left" nowrap="nowrap">
```

<%-- The following block builds the menu in the table cell. It first adds an unordered list to the cell and sets its style class. Then, an IF statement checks to see if the book is in VIEW mode. If true, CSS styles are put in the request as attributes to be used by the menu.

After the attributes are added to the request, the skeleton's submenu.jsp is inserted, which does the following:

- * Gets the CSS styles from the request.
- * Gets the book's child pages and books.

- * Creates list items `` of the children and creates links out of them.

The `menuHookClass` at the end of the block is used by the skin's `menu.js` file to insert the rendered menu. The `` that is generated is a menu structure description that is read and rewritten by `menu.js`.

- * Adds CSS styles to the request and includes `submenu.jsp` to handle the menus of nested books.

After the menu is built, the CSS styles are removed from the request.

--%>

```
<ul
  <render:writeAttribute name="id" value="<%=
    menu.getPresentationId() %>"/>
  <render:writeAttribute name="class" value="<%=
    menu.getPresentationClass() %>" defaultValue="<%=
    menuClass %>"/>
  <render:writeAttribute name="style" value="<%=
    menu.getPresentationStyle() %>"/>
><%
  if (book.getWindowMode().equals(WindowCapabilities.VIEW))
  {
    request.setAttribute(BookPresentationContext.class.getName() + ".root-flag",
Boolean.TRUE);
    request.setAttribute(BookPresentationContext.class.getName() + ".menu-item",
book);
    request.setAttribute(BookPresentationContext.class.getName() + ".menu-class",
menuClass);
```



```

        request.setAttribute(BookPresentationContext.class.getName() +
".menu-item-class", menuItemClass);
        request.setAttribute(BookPresentationContext.class.getName() +
".menu-item-active-class", menuItemActiveClass);
        request.setAttribute(BookPresentationContext.class.getName() +
".menu-item-link-class", menuItemLinkClass);
        %><jsp:include page="submenu.jsp"/><%
        request.removeAttribute(BookPresentationContext.class.getName() +
".root-flag");
        request.removeAttribute(BookPresentationContext.class.getName() +
".menu-item");
        request.removeAttribute(BookPresentationContext.class.getName() +
".menu-class");
        request.removeAttribute(BookPresentationContext.class.getName() +
".menu-item-class");
        request.removeAttribute(BookPresentationContext.class.getName() +
".menu-item-active-class");
        request.removeAttribute(BookPresentationContext.class.getName() +
".menu-item-link-class");
    }
    %></ul>
    <div class="<%= menuHookClass %>"></div>
</td>
<%-- The following block adds a table cell next to the menu table cell
    if a menu is present. The <render:endRender> contents are inserted
    in the HTML after all menu children are inserted, which closes
    the menu table.
--%>

<%

```

```

        if (menuChildren != null && menuChildren.size() > 0)
        {
%>
            <td class="<%= menuButtonsClass %>" align="right" nowrap="nowrap">

<%
        }
%>
</render:beginRender>
<render:endRender>
<%
        if (menuChildren != null && menuChildren.size() > 0)
        {
%>
            </td>

<%
        }
%>
        </tr>
    </table>
</div>
<%-- End Multi Level Menu --%>
</render:endRender>

```

submenu.jsp

The submenu.jsp is inserted inside the multilevelmenu.jsp. It retrieves a book's child books and pages and builds the navigation links to those children. Comments are shown in bold text.

```

<%@ page import="java.util.Iterator,
        java.util.List,
        com.bea.netuix.servlets.controls.page.BookPresentationContext,

```

```

        com.bea.portlet.PageURL,
        com.bea.netuix.servlets.controls.page.PagePresentationContext"%>
<%@ page session="false"%>
<%-- The following block gets the CSS styles placed in the request by
      multilevelmenu.jsp.
--%>

<%
    Boolean isRoot
        = (Boolean) request.getAttribute(BookPresentationContext.class.getName() + ".root-flag");
    BookPresentationContext bookCtx
        = (BookPresentationContext)
        request.getAttribute(BookPresentationContext.class.getName() +
        ".menu-item");
    String menuClass
        = (String) request.getAttribute(BookPresentationContext.class.getName() +
        ".menu-class");
    String menuItemClass
        = (String) request.getAttribute(BookPresentationContext.class.getName() +
        ".menu-item-class");
    String menuItemActiveClass
        = (String) request.getAttribute(BookPresentationContext.class.getName() +
        ".menu-item-active-class");
    String menuItemLinkClass
        = (String) request.getAttribute(BookPresentationContext.class.getName() +
        ".menu-item-link-class");
%>

```

<%-- The following block checks to see if the book and its children are visible. If true, the labels of the children are retrieved, iterated over, and inserted as hyperlinked list items inside the unordered list inserted by multilevelmenu.jsp. Notice the nested at the end of the block, which provides for submenu nesting.

```
--%>

<%
    if (!bookCtx.isHidden() && bookCtx.isVisible())
    {
        if (bookCtx instanceof BookPresentationContext)
        {
            List bookChildren = bookCtx.getPagePresentationContexts();
            Iterator it = bookChildren.iterator();
            while (it.hasNext())
            {
                PagePresentationContext childPageCtx = (PagePresentationContext)
                    it.next();
                if (!childPageCtx.isHidden() && childPageCtx.isVisible())
                {
                    %><li class="<%= isRoot.booleanValue() &&
                        childPageCtx.isActive() ? menuItemActiveClass : menuItemClass
                    %>"><%
                        %><a class="<%= menuItemLinkClass %>" href="<%=
PageURL.createPageURL(request, response, childPageCtx.getDefinitionLabel()).toString()
                    %>"><%= childPageCtx.getTitle() %></a><%
                        if (childPageCtx instanceof BookPresentationContext)
```

```

    {
        %><ul class="<%= menuClass %>"><%
            request.setAttribute(BookPresentationContext.class.getName()
                + ".root-flag", Boolean.FALSE);
            request.setAttribute(BookPresentationContext.class.getName()
                + ".menu-item", childPageCtx);
            %><jsp:include page="submenu.jsp"/><%
            request.removeAttribute(BookPresentationContext.class.getName() + ".root-flag");
            request.removeAttribute(BookPresentationContext.class.getName() +
".menu-item");
            %></ul><%
        }
        %></li><%
    }
}
}
}
}
%>

```

JavaScript in Menus

The menus in a desktop use JavaScript functions for such functionality as drop-down menus and rollovers. These JavaScript functions are called from the skeleton's `body.jsp`, which contains the following entry:

```
<render:writeAttribute name="onload" value="<%= body.getOnloadScript() %>" />
```

The `onload` value is retrieved from the following property in the skin's `skin.properties` file:

```
document.body.onload: initSkin()
```

Following is the HTML written by the body.jsp:

```
<body  
  
    class="bea-portal-body"  
  
    onload="initSkin();"   
  
>
```

The `initSkin()` JavaScript function is the base function that calls menu-rendering functions in other JavaScript files. The `initSkin()` function is contained in the `skin.js` file. Other menu functions are contained in the `menu.js` and `menufx.js` files. Since all of those JavaScript files are listed in the skin's `skin.properties` file, they are automatically added to the HTML `<head>` region at rendering, and the functions they contain are recognized.

The next section describes the final process of the skeleton JSPs and `skin.properties` being converted to HTML.

3. JSP skeleton files and `skin.properties` are rendered as HTML

The previous section described the skeleton JSPs that are used to convert a book with a multi-level menu to HTML. The descriptions in that section described briefly some of the HTML generated by the JSPs.

This section shows the final HTML that is generated for a book, describes where it came from, and shows where some of the CSS styles used are defined.

Not all HTML for the desktop is shown in the following table. Only the sections that relate to the look & feel and the example book are shown.

`skin.properties` and `skin_custom.properties` - The paths to skeletons, skins, images, style sheets, and JavaScript files in the HTML `<head>` region are inserted from the skin's `skin.properties` and `skin_custom.properties` files. To see the original `skin.properties` entries, see [The skin.properties](#)

File in "How Look & Feel Determines Rendering." The <head> tag is inserted by the head.jsp file used for the shell. The <title> is inserted from the desktop title in the .portal file.

The first three <meta> tags are for testing and debugging purposes. These can be removed from skin.properties by setting the enable.meta.info property to false.

```
<head>
```

```
<title>New Portal Desktop</title>
```

```
<meta name="bea-portal-meta-skeleton" content="/framework/skeletons/default"/>
```

```
<meta name="bea-portal-meta-skin" content="/framework/skins/avitek"/>
```

```
<meta name="bea-portal-meta-skin-images" content="/framework/skins/avitek/images"/>
```

```
<link href="/sampleportal/framework/skins/avitek/css/body.css" rel="stylesheet"/>
```

```
<link href="/sampleportal/framework/skins/avitek/css/button.css" rel="stylesheet"/>
```

```
<link href="/sampleportal/framework/skins/avitek/css/window.css" rel="stylesheet"/>
```

```
<link href="/sampleportal/framework/skins/avitek/css/plain/window.css" rel="stylesheet"/>
```

```
<link href="/sampleportal/framework/skins/avitek/css/portlet.css" rel="stylesheet"/>
```

```
<link href="/sampleportal/framework/skins/avitek/css/book.css" rel="stylesheet"/>
```

```
<link href="/sampleportal/framework/skins/avitek/css/fix.css" rel="stylesheet"/>
```

```
<link href="/sampleportal/framework/skins/avitek/css/layout.css" rel="stylesheet"/>
```

```
<link href="/sampleportal/framework/skins/avitek/css/form.css" rel="stylesheet"/>
```

```
<script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/menu.js"></script>
```

```
<script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/util.js"></script>
```

```
<script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/delete.js"></script>
```

```
<script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/float.js"></script>
```

```
<script type="text/javascript"
```

```
src="/sampleportal/framework/skins/avitek/js/menufx.js"></script>
```

```
<script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/skin.js"></script>
```

```
</head>
```

The following section shows the HTML that is produced by each skeleton JSP.

book.jsp

```
<div
    class="bea-portal-book-invisible"
```

```
>
```

multilevelmenu.jsp

```
<div class="bea-portal-ie-table-buffer-div">
    <table border="0" cellpadding="0" cellspacing="0" width="100%">
        <tr>
            <td class="bea-portal-book-menu-container" align="left"
                nowrap="nowrap">
                <ul
                    class="bea-portal-book-menu"
                >
```

submenu.jsp

```
<li class="bea-portal-book-menu-item-active"><a class="bea-portal-book-menu-item-link"
href="http://localhost:7001/sampleportal/my.portal?_nfpb=true&_pageLabel=my_page_6">Ne
w Page</a></li>
```

```
<div
    class="bea-portal-book-content"
>
```

```
<%-- The book content (sub-books and pages) is inserted here. -->
```

```
</div>
```

```
</div>                </ul>
```

```
<div class="bea-portal-book-menu-hook"></div>
```

```
</td>
```



```

        </tr>
    </table>
</div>

```

When the desktop for this example is rendered, the following appears in the browser:

Rendered desktop

The circled area in this figure is the only content rendered for the book. The book contains only one page, so there is only one menu item for the book. The "Page 1" and "My Book" tabs are menu items rendered by the parent Main Page Book. That is why you do not see the "My Book" in the previous HTML block: because the book is responsible for rendering only a menu of its child books and pages.

If "New Page" contained a portlet, the portlet would appear in the browser. However, the rendering of the page and portlet is handled by different skeleton JSPs: one to provide a container for the page content, one to render the layout of the page (table cells that contain portlets and sub-books), and a few to handle the rendering of the portlet.

The book is responsible for rendering only two things:

The menu of sub-books and pages it contains.

Opening and closing `<div>` tags to serve as the container for sub-books, pages, portlets, and other sub-components contained in the book.

CSS Styles in the Example

As you can see from the previous example of rendered HTML code, the skeleton JSPs insert many CSS styles. For example, the `multilevelmenu.jsp` inserts

```
<td class="bea-portal-book-menu-container" ...>
```

The style classes inserted by `multilevelmenu.jsp` are rewritten by the skin's `menu.js` file.

Also, some of the style classes inserted by the skeleton JSPs are not defined in any of the CSS files provided by BEA. You can add these style classes to your custom CSS files to control those styles in your portal desktops.

To determine which styles you want to modify, see [The Look & Feel Editor](#).

Changing Look & Feel

If the look & feel is changed, a different skin and skeleton is referenced by the look & feel file, and rendering is subject to that skin and skeleton. With a different skin and skeleton, CSS files and script code can change completely.

Summary

There are three basic stages in the portal rendering process: building a portal in XML, portal XML components being mapped to skeleton JSPs, and skeleton JSPs rendering the portal desktop in HTML. The latter two stages are handled automatically by the portal framework.

There is a rendering difference between viewing a portal desktop in development mode and in administration/end user mode. In development mode, when you view the .portal file in a browser you see it in "single file" mode, meaning the desktop is being rendered from the file system. In administration/end user mode, you view a portal desktop in a browser in "streamed" mode, meaning the desktop components are being streamed from a database. When you create a portal desktop in the WebLogic Administration Portal using a .portal file as a template for the desktop, the portal components are added to the database and are decoupled from the original .portal file.

Creating a New Style

Creating a New Style

Working with Skins

Modifying the Title Bar

Modifying Fonts

Adding a Style

Additional Exercises

These will be workbook-like exercises—“whiz-bang examples”—for which we will not provide steps, just an “assignment.” The solutions will be contained in the tutorial folder

Changing Icons

Working with Skins: Best Practices

Working with Skins

Modifying Skeletons

Adding Text to a Title Bar

Adding a Title Bar

Adding a Subtitle Bar

Additional Exercises

Adding a Subtitle Bar

Modifying Skeletons

Working with an Existing Look-and-Feel

Creating the Element

Adding the Element to the Look-and-Feel

Adding a Behavior

Overriding and Extending a Deployed Look-and-Feel

Additional Exercises

Working with an Existing Look-and-Feel

Integrating Portlet Content Styles with a Look and Feel

Overview

Linking Dependencies to a Portlet

Overriding Skins

Integrating Portlet Content Styles with a Look and Feel

Working with Themes, Classifications, and Localizations

Overview

Configuring Themes, Classifications, and Localizations in a New Config File

Specializations

Matching Client Classes

Working with Themes, Classifications, and Localizations

Working with Genes

Overview

What are Genes

How Genes Work

Exercise: Using Genes to Change a Color Set

Integrating Third Party Multi-level Menus

Overview

Impact on Control Tree Rendering

Exercise: Integrating Third-Party Multi-Level Menus

Integrating Third Party Multi-level Menus

UI Design Resources

UI Design Resources