BEA WebLogic Server®

Extending the Administration Console

Version 9.1
Revised: December 16, 2005

Copyright

Restricted Rights Legend

Trademarks and Service Marks

Contents

3. Setting Up a Development Environment

4. Creating a Message Bundle

5. Rebranding the Administration Console

6. Adding Portlets and Navigation Controls

7. Using BEA Templates and JSP Tags

8. Archiving and Deploying Console Extensions

CHAPTER 1

Introduction and Roadmap

Administration Console extensions enable you to add content to the WebLogic Server Administration Console, replace content, and change the logos, styles and colors without modifying the files that are installed with WebLogic Server. For example, you can add content that provides custom monitoring and management facilities for your applications.

The Administration Console is a J2EE Web application that uses the WebLogic Portal framework, Apache Beehive, Apache Struts, Java Server Pages (JSP), and other standard technologies to render its user interface (UI) and content. It also uses the WebLogic Portal framework to enable extensions.

The following sections describe the contents and organization of this guide—*Extending the Administration Console*.

Document Scope and Audience

This document is a resource for software vendors who embed or rebrand WebLogic Server in their products, software vendors who develop security providers or other resources that extend the functionality of WebLogic Server, and J2EE application developers who want to provide custom monitoring and configuration features for their applications.

It is assumed that the reader is already familiar with using Java, JavaServer Pages, and Apache Struts or Apache Beehive to develop J2EE Web applications. This document emphasizes a hands-on approach to developing a limited but useful Administration Console extension. For information on applying Administration Console extensions to a broader set of management problems, refer to documents listed in "Related Documentation" on page 1-3.

Guide to this Document

- This chapter, Introduction and Roadmap, introduces the organization of this guide.
- Chapter 2, "Understanding Administration Console Extensions," introduces the building blocks for creating Administration Console extensions.
- Chapter 3, "Setting Up a Development Environment," describes how to set up your environment for developing Administration Console extensions.
- Chapter 4, "Creating a Message Bundle," describes how to encapsulate the text that your extension displays into properties files that can be localized.
- Chapter 5, "Rebranding the Administration Console," describes how to create a WebLogic Portal Look and Feel and deploy it as an Administration Console extension.
- Chapter 6, "Adding Portlets and Navigation Controls," describes how to add portlets that contain simple, static content to the Administration Console.
- Chapter 7, "Using BEA Templates and JSP Tags," describes how to create an extension that uses the Administration Console's JSP templates, styles, and JSP tag library.
- Chapter 8, "Archiving and Deploying Console Extensions," describes how deploy your extension.

Related Documentation

This section provides links to documentation that describes the technologies used by the Administration Console. The more you understand these technologies, the more complex extensions you can create.

Because the Administration Console uses the WebLogic Portal® framework to render its user interface, the process of extending the Administration Console is similar to creating or editing an existing WebLogic Portal application. For information on the WebLogic Portal framework, see:

- "How Do the WebLogic Portal Framework and WebLogic Portal Differ?" on page 2-2
- *Portal User Interface Framework Guide* at http://e-docs.bea.com/wlp/docs81/lookandfeel/index.html
- *White Paper: WebLogic Portal Framework* at http://e-docs.bea.com/wlp/docs81/whitepapers/netix/index.html

For information on JavaServer Pages, see *JavaServer Pages Technology* at http://java.sun.com/products/jsp/index.jsp.

For information on Apache Struts, see *The Apache Struts Web Application Framework* at http://struts.apache.org/.

For information on Apache Beehive, see http://beehive.apache.org/.

New and Changed Console-Extension Features in This Release

This release contains no changes for Administration Console extensions.

Changes to Console-Extension Features in the WebLogic Server 9.0 Release

WebLogic Server 9.0 completely re-designed the WebLogic Server Administration Console. Because the new architecture is so different, WebLogic Administration Console extensions built for releases prior to WebLogic Server 9.0 will not function in 9.0 or later. See Introduction and Roadmap in *Extending the Administration Console* for WebLogic Server 9.0.

CHAPTER 2

Understanding Administration Console Extensions

The following sections describe Administration Console extensions:

What Is an Administration Console Extension?

An Administration Console extension is a JAR file that contains the resources for a section of a WebLogic Portal Web application. When you deploy the extension, the Administration Console creates an in-memory union of the files and directories in its WAR file with the files and directories in the extension JAR file. Once the extension has been deployed, it is a full member of the Administration Console: it is secured by the WebLogic Server security realm, it can navigate to other sections of the Administration Console, and if the extension modifies WebLogic Server resources, it participates in the change control process.

The simplest extension adds content to the Administration Console's home page (desktop). The JAR file for such an extension contains:

- A NetUI Extension XML file that describes the location in the UI in which you want your extension to display.
- An XML file that defines a WebLogic Portal portlet, which is a container for JSPs and other types of content.
- A JSP file that contains the content you want to display.

The JAR file for more complex extensions can contain any of the following additional resources:

  – If the extension displays content in tabs within the Administration Console UI, the JAR contains XML files that describe other types of WebLogic Portal UI controls, such as tabs and subtabs (see "Extension Points in the Administration Console" on page 2-3).

  – If the extension uses Apache Struts to encapsulate business logic and navigation logic, the JAR file contains configuration files and Java classes for Apache Struts applications.

  – If the extension uses Apache Beehive to encapsulate business logic and navigation logic, the JAR file contains configuration files and Java classes for Apache Beehive applications.

  – Java classes, image files, or other types of resources that can be used in J2EE Web applications.

Note: The Administration Console does not support WSRP portlets or portlets based on JSR 168.

How Do the WebLogic Portal Framework and WebLogic Portal Differ?

The WebLogic Portal framework provides basic support for rendering the UI. The full WebLogic Portal product provides the framework and additional features such as personalization, interaction management, content management, and the ability for end users to customize their portal desktops.

If your BEA product license includes only WebLogic Server, then you can use the WebLogic Portal framework when creating Administration Console extensions. If you want your own Web applications to provide a portal interface, you can purchase the WebLogic Portal product.

Extension Points in the Administration Console

An extension point is a location in the Administration Console UI at which you can add or replace content. The UI for the Administration Console is rendered by groups of specialized WebLogic Portal components called UI controls. Each group of controls is responsible for rendering a specific part of the UI. For example, one group renders the two-column layout that you see after you log in to the Administration Console. Other groups render individual tabs in the tabbed interface.

The Administration Console attaches unique labels to many of its UI controls, and each labeled control is an extension point. You can also use these labels with WebLogic Server JSP tags to forward requests to specific UI controls. If a UI control is not identified by a label, you cannot extend it or forward to it. You must either interact with its labeled ancestor control or a labeled child control.

Hierarchy of UI Controls

UI controls for an application are defined in an XML file called a portal include file (`.pinc` file). The schema for this XML file specifies a hierarchy of UI controls, but some UI controls can be used at multiple levels in the hierarchy. The following list describes the types of UI controls that you will encounter most frequently while developing extensions (see Figure 2-1):

- Desktop

  The top level of the UI control hierarchy. It contains the Look and Feel for the Administration Console and the top-level book control.

- Look and Feel

  A collection of images, cascading style sheets, XML files, and other file types that control the appearance of a portal application.

- Book

  Aggregates a set of pages or other books. It can contain an optional menu control that provides navigation among its pages and books. Many books in the Administration Console use this menu control to render tabs, such as the domain's Configuration: General tab.

- Page

  Contains a layout, portlets, or books.

- Layout

  Defines a grid in the UI. Each column in the grid is called a placeholder, and each placeholder can host zero or more portlets or books.

  Most pages in the Administration Console use a single column layout, but one of the top pages uses a two-column layout to create the left column that contains the Change Center, Domain Structure, and other portlets, and the right column that contains the tabbed interface.

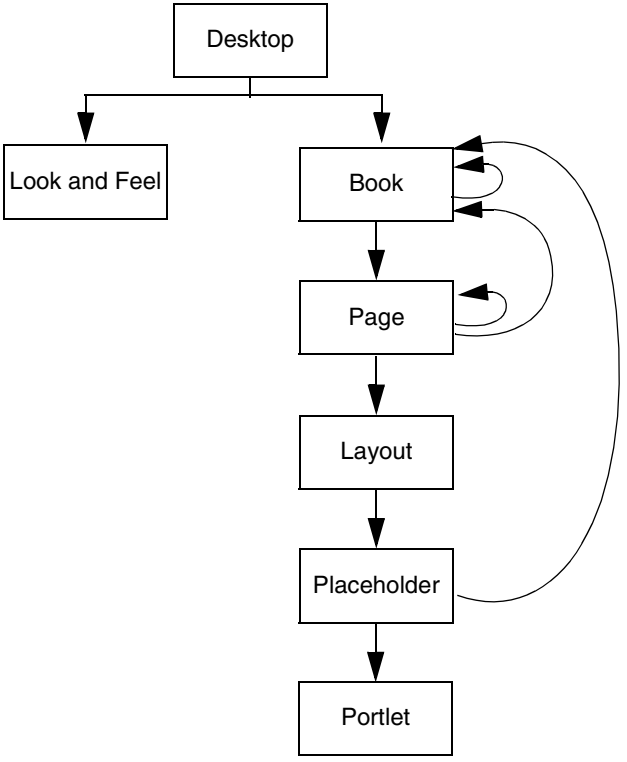
- Portlet

  Defines static and dynamic content to display. You can add portlets to the Administration Console that contain JSP files or that forward to Struts `Actions` or Beehive Page Flows.

For information about the schema for UI controls, see *Portal Support Schema Reference*.

Note: Figure 2-1 omits some intermediate controls in the hierarchy for the sake of brevity. For example, a book control does not directly contain a page control. Instead, a book contains a control named **content**, and the content control contains the page control.

Figure 2-1 Subset of the UI Control Hierarchy



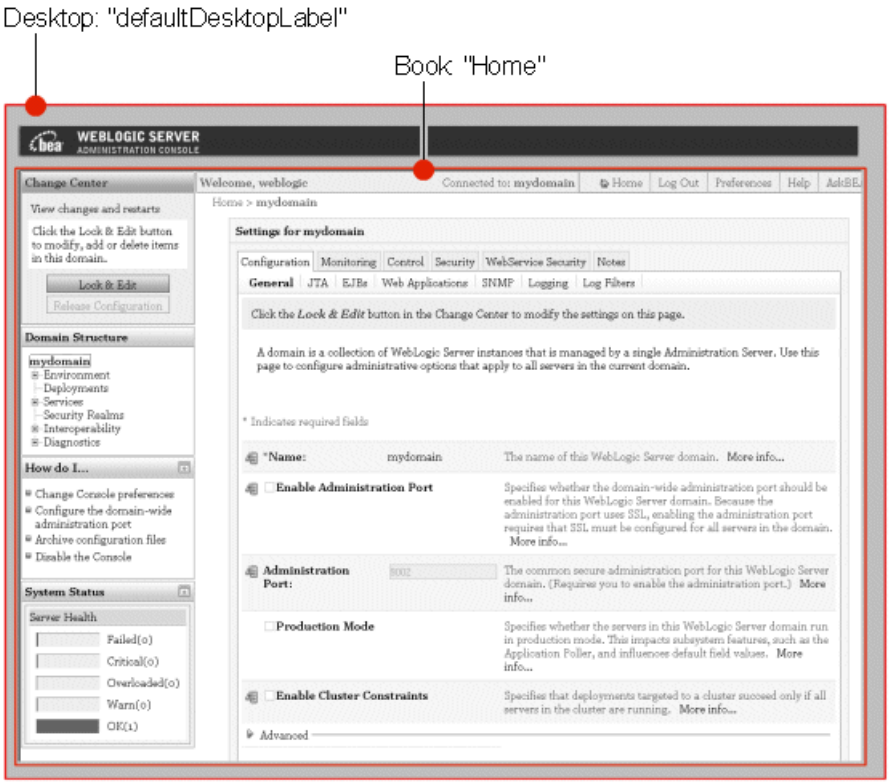
The following sections describe the extension points in the Administration Console:

- "The Administration Console Desktop" on page 2-6
- "The Home Book and Page" on page 2-8
- "The ContentBook" on page 2-9
- "Summary of the Administration Console UI Controls" on page 2-11

The Administration Console Desktop

Every WebLogic Portal Web application must have at least one desktop control, and the Administration Console supports **only** one. Its label is `defaultDesktopLabel` (see Figure 2-2).

Figure 2-2 The Desktop



Extending the Desktop

The only type of extension that is supported at this level of the Administration Console is a Look and Feel extension, which replaces BEA's logos, colors, and fonts with yours. See "Rebranding the Administration Console" on page 5-1.

You cannot replace the Home book or add other controls to the desktop.

The Administration Console Look and Feel

The Look and Feel for the Administration Console defines the fonts and colors, BEA logos, the layout of portal components, and the navigation menus.

Note: Because the Administration Console uses only the WebLogic Portal Framework, it supports only a single Look and Feel. Portal applications that use the entire set of features available with a license for the BEA WebLogic Portal product can support multiple Look and Feels that are personalized based on user or group ID.

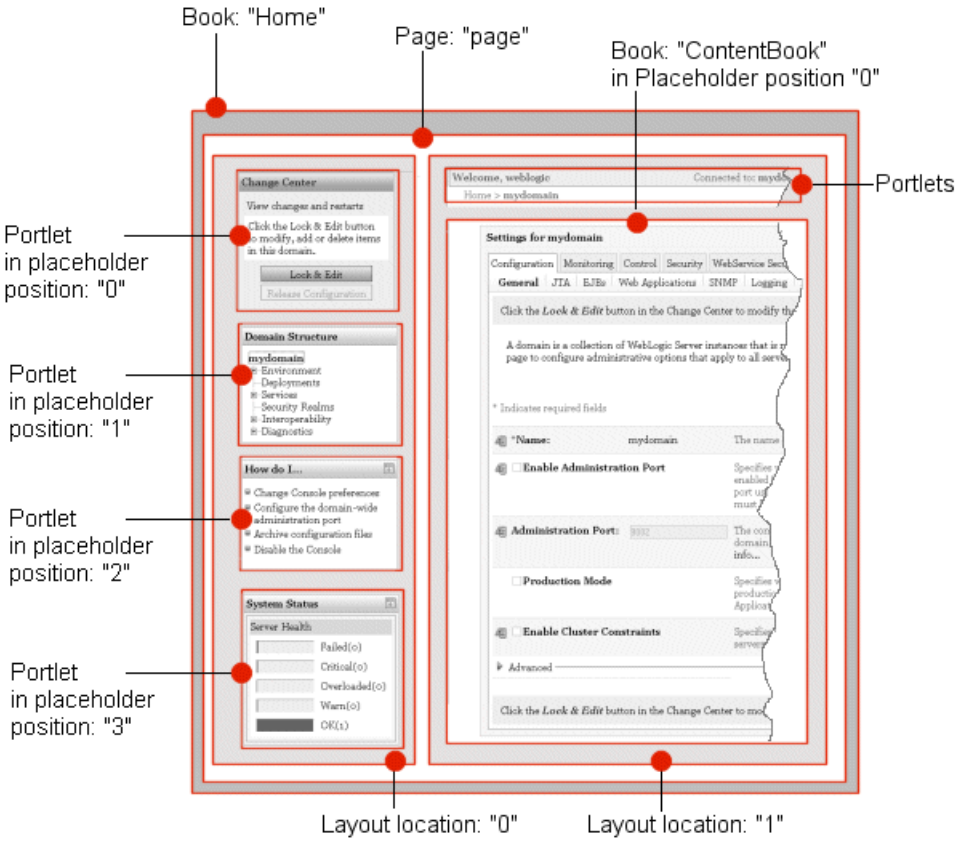
Extending the Look and Feel

Creating a simple Look and Feel extension that contains your company's logos, fonts, and color scheme requires you to copy a sample Look and Feel that WebLogic Server provides and then replace the logos and some cascading style sheet (CSS) definitions. Making complex changes to the WebLogic Server Look and Feel, such as changing the layout of portal components and navigation menus, requires an advanced knowledge of WebLogic Portal Look and Feels. If you have a license for WebLogic Workshop, you can use its Look and Feel editor to make these complex changes. For more information about Look and Feels, see the *Portal User Interface Framework Guide*.

The Home Book and Page

The top-level book in the Administration Console is identified by the label `Home`. It contains a single page (labeled `page`) within which resides all of the Administration Console content (see Figure 2-3).

Figure 2-3 The Home Book and Page



The `page` page uses a two-column layout. The left column (layout location 0) contains portlets that provide essential services when using the Administration Console. The right column (layout location 1) contains:

- Portlets:
  - The topmost portlet displays a welcome message and contains buttons that launch online help and other services.
  - The second portlet displays breadcrumbs, which are a series of hypertext links that keep a history of your navigation in the Administration Console.
  - A third portlet is hidden by default and displays error messages and other status messages.
- A book named `ContentBook`. See "The ContentBook" on page 2-9.

Extending the Home Book

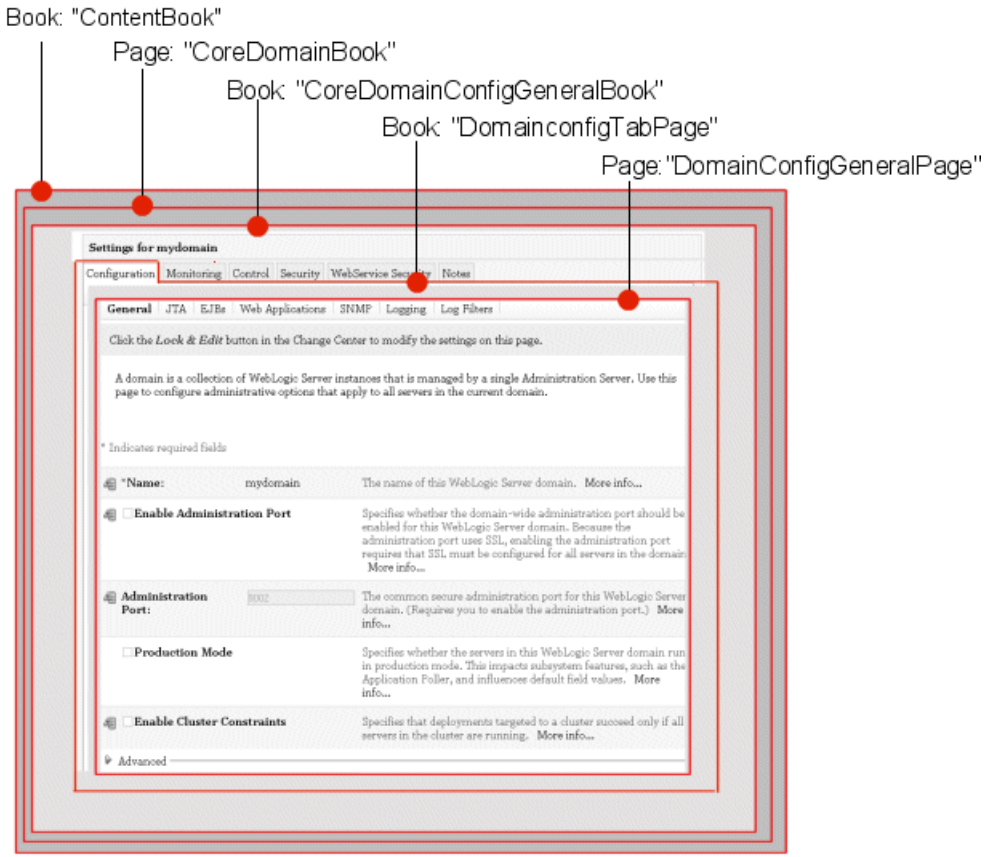
The simplest extensions within the `Home` book add portlets to either column of its `page` page. For example, below the System Status portlet, you can add a portlet that monitors your applications.

More complex extensions can append a book to the `Home` book. Such an extension causes the `Home` book and the extension book to display as two separate tabs.

The ContentBook

The `ContentBook` is a book that contains over 40 pages (see Figure 2-4), but it displays only one page at a time. Navigational controls throughout the Administration Console determine which page is displayed.

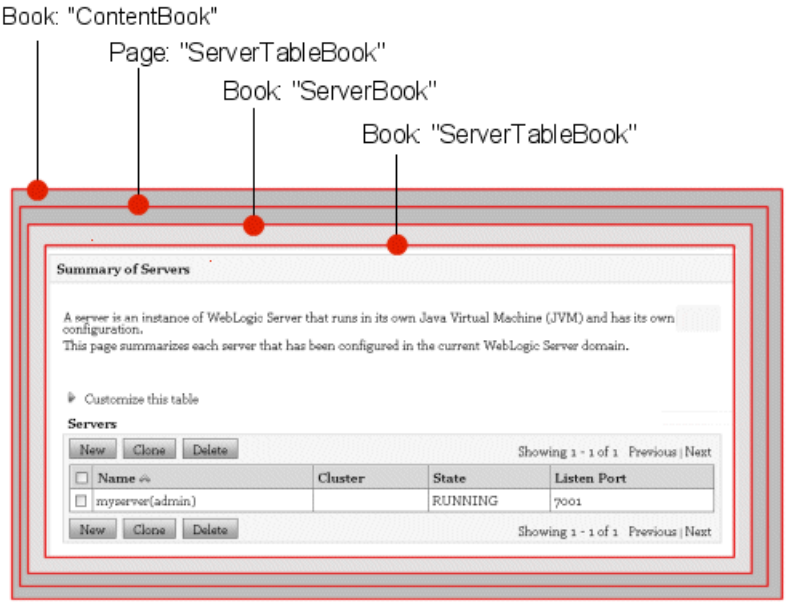
Figure 2-4 The ContentBook



In Figure 2-4, a page named `CoreDomainBook` contains a book named `CoreDomainConfigGeneralBook`. The `CoreDomainConfigGeneralBook` contains six child books and a special UI control named `singleLevelMenu` that renders a tab for each child book (Configuration, Monitoring, Control, Security, WebService Security, and Notes). In turn, each child book (such as `DomainconfigTabPage`) contains several child page controls and the `singleLevelMenu` control. The Look and Feel causes the `singleLevelMenu` control to generate subtabs for the page controls at this level (General, JTA, EJBs, Web Applications, SNMP, Logging, and Log Filters).

Some content-specific books do not display a tabbed interface for their child books. Figure 2-5 shows the `ServerBook`, which does not display a tabbed interface.

Figure 2-5 ServerTableBook



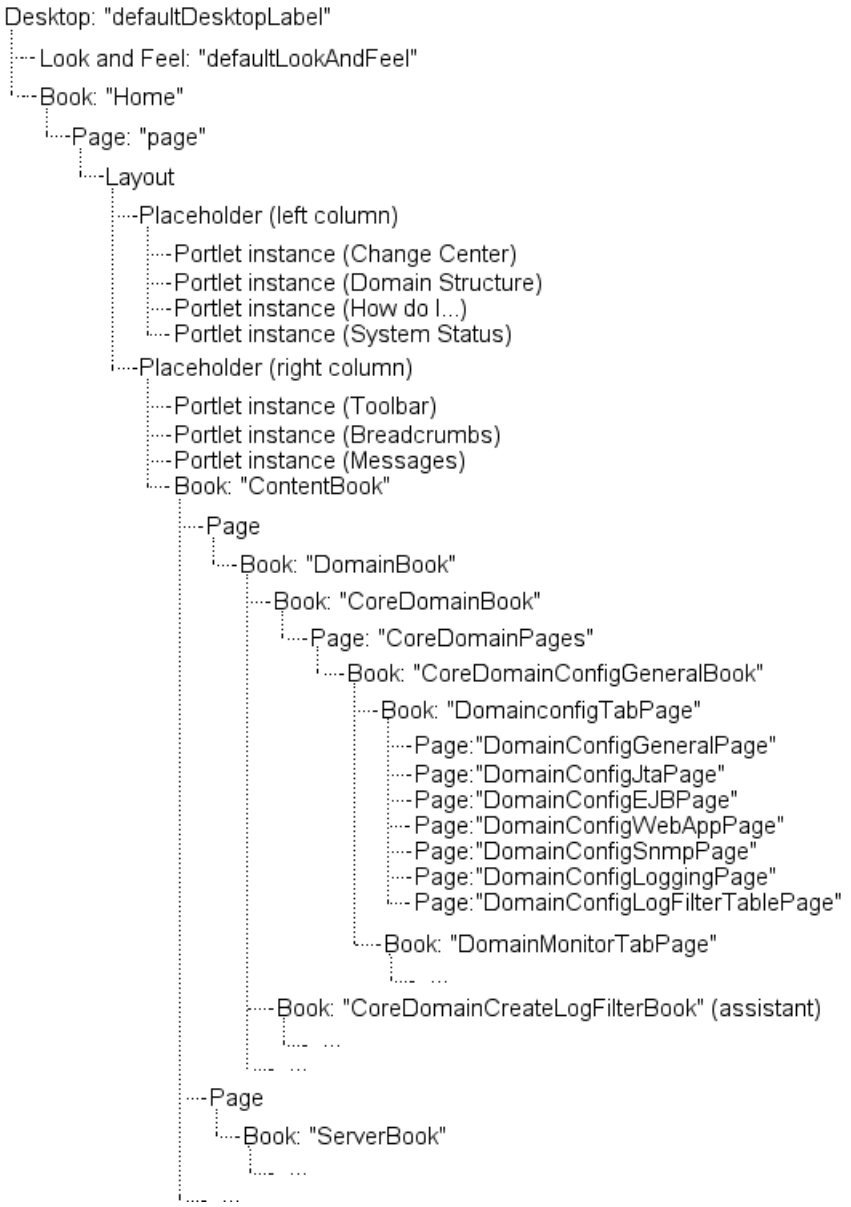
Extending the ContentBook

The simplest extensions within the `ContentBook` add a child book to create a tab in a content-specific book or add a child page to create a subtab. See "Define UI Controls (Optional)" on page 6-12.

Summary of the Administration Console UI Controls

Figure 2-6 shows the top levels of the Administration Console's labeled UI controls. For a complete list of labeled UI controls, including all of the content-specific books, download and install a Look and Feel extension that causes the Administration Console to display labels for its controls. See "Deploy a Development Look and Feel to See UI Control Labels" on page 3-5.

Figure 2-6 Summary of the UI Control Hierarchy



JSP Templates and Tag Libraries

BEA provides JSP templates and tab libraries that you can use to render such UI features as tables, data-entry boxes, and buttons. For information about the JSP templates, see "WebLogic Server JSP Templates" on page 7-18.

JSP Tag Libraries

For each of the tag libraries in Table 2-1, the Administration Console provides runtime support by default. If you want development support for these libraries (for example, if you use an integrated development environment that provides code completion for JSP tags), you must configure your development environment to include these tags. (See "Setting Up a Development Environment" on page 3-1.)

Note: You can create custom tag libraries or use additional tag libraries, but you must include all of the necessary support files for custom tag libraries in your extension JAR file. See *Programming WebLogic JSP Tag Extensions*.

Table 2-1 Included JSP Tag Library Support

Tag Library	Contains
`console-html.tld`	WebLogic Server JSP tags for creating HTML forms and tables that match the functionality of the forms and tables in the Administration Console. Use these tags only to extend the WebLogic Server Administration Console. The documentation for this tag library is in the *WebLogic Server JSP Tags Reference*.
`render.tld`	Convenience tag for generating a portal framework URL. See `<render:pageUrl> Tag` in *WebLogic Workshop Help*.

`beehive-netui-tags-template.tld`
`beehive-netui-tags-databinding.tld`
`beehive-netui-tags-html.tld`
	Apache Beehive JSP tags for associating JSPs with a JSP template, binding data, and generating basic HTML tags.
	You can download the Beehive distribution, which includes the tag libraries and documentation from http://beehive.apache.org/downloads.html.

`c.tld` `fmt.tld`	JavaServer Pages Standard Tag Library (JSTL) tags which provide core functionality common to many JSP applications. You can download the JSTL distribution from http://java.sun.com/products/jsp/jstl/downloads/index.html. The documentation for these tag libraries is in the *JSTL Tag Library Reference*.

`struts-bean.tld`
`struts-html.tld`
`struts-logic.tld`
`struts-nested.tld`
`struts-template.tld`
`struts-tiles.tld`
	Apache Struts tags for interacting with the Struts framework.
	You can download the Struts distribution from http://struts.apache.org/download.cgi.
	The documentation for these tag libraries is available from http://struts.apache.org/.

Example: How Struts Portlets Display Content

The following steps describe how the portal framework uses an extension's source files to find and display a Struts portlet as a tab in `ContentBook` (see Figure 2-7):

1. The portal framework starts by parsing the extension's `netuix-extension.xml` file.

   The `netuix-extension.xml` file in this example specifies that the portal framework should load a `.pinc` file named `medrecMonitor.pinc` and display its contents as a child of the `CoreDomainConfigGeneralBook` book:

   ```
   <book-extension>
      <book-location>
         <parent-label-location label="CoreDomainConfigGeneralBook"/>
         <book-insertion-point action="append"/>
      </book-location>
      <book-content content-uri="/controls/medrecMonitor.pinc"/>
   </book-extension>
   ```

2. The portal framework loads the `medrecMonitor.pinc` file, which defines a page UI control and specifies that the page contains a portlet:

   ```
   <netuix:page markupName="page" markupType="Page">
   ...
      <netuix:portletInstance markupType="Portlet"
         instanceLabel="medrecMonitor.Tab.Portlet"
         contentUri="/portlets/medrec_monitor_tab.portlet"/>
   </netuix:page>
   ```

3. The portal framework loads the portlet file, which names a Struts `Action` to run:

   ```
   <portal:root>
      <netuix:portlet
         definitionLabel="MyPortlet"
         title="my.portlet.title">
         <netuix:strutsContent module="/medrecMBean"
            action="MedRecMBeanFormAction"
            refreshAction="MedRecMBeanFormAction"/>
      </netuix:portlet>
   </portal:root>
   ```

   In the `netuix:strutsContent` element, the `module="/medrecMBean"` attribute indicates that the definition for the `RetrieveCustomMBeansAction` Struts `Action` is located in the Struts configuration file for the Struts module named `medrecMBean`. The Struts naming convention requires that this configuration file be named `struts-auto-config-medrecMBean.xml`.

4. The portal framework hands control to the Struts controller servlet, which parses the `struts-auto-config-medrecMBean.xml` file and finds the following definition for the `RetrieveCustomMBeansAction`:

```
<action path="/MedRecMBeanFormAction"
   type="com.bea.medrec.extension.MedRecMBeanFormAction"
   name="medrecMBeanEJBForm"
   scope="request"
   validate="false">
   <forward name="success" contextRelative="true"
      path="/ext_jsp/form_view.jsp"/>
   </action>
```

5. When the Struts controller encounters the `name="medrecMBeanEJBForm"` attribute of the `action` element, it looks in the same Struts configuration file for the definition of a form bean that is named `medrecMBeanEJBForm`.

   When it finds the following element in configuration file:

```
<form-bean name="medrecMBeanEJBForm"
   type="org.apache.struts.action.DynaActionForm">
   <form-property name="name"
      type="java.lang.String"/>
   <form-property name="handle"
      type="com.bea.console.handles.Handle"/>
   <form-property name="totalRx"
      type="java.lang.Integer"/>
</form-bean>
```

   it initializes a Java bean of type `org.apache.struts.action.DynaActionForm` with properties named `name`, `handle`, and `totalRx`.

6. The Struts controller invokes the `com.bea.medrec.extension.MedRecMBeanFormAction` class and passes to this class the `DynaActionForm` bean that it instantiated.

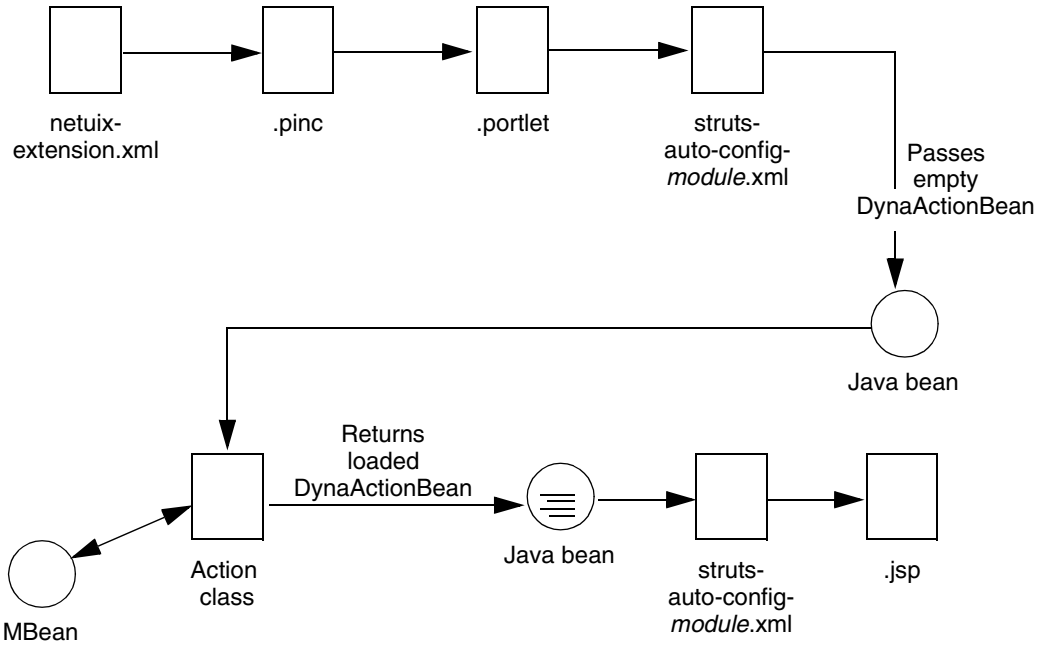
7. The `MedRecMBeanFormAction` class gathers data from an MBean in the MedRec application and populates the properties in the `DynaActionForm` bean with data from the MedRec MBean.

   The `MedRecMBeanFormAction` class returns the populated `DynaActionForm` bean.

8. The Struts controller serializes the `DynaActionForm` bean, sets it in an HTTP request, and then forwards to a JSP.

9. The JSP uses JSP tags to display data in the `DynaActionForm` bean.

Figure 2-7 Overview of Loading a Struts Portlet



CHAPTER 3

Setting Up a Development Environment

BEA provides all of the JSP tag libraries, schemas, and base Java classes that you need to develop a console extension. Because an Administration Console extension is a collection of XML files, Java classes, JSPs, and other standard Web-related resources, you can use any text editor or Integrated Development Environment to develop your extension.

The following sections describe setting up an environment for developing Administration Console extensions:

Set Up the Classpath (Optional)

If you are creating Apache Struts classes or Beehive Page Flow classes for your extension, you need a set of Apache classes in your classpath. If you are adding nodes to the NavTreePortlet, you need a set of BEA classes.

To add these classes to your classpath, run the following script:
`WL_HOME\server\bin\setWLSEnv.cmd` (or `setWLSEnv.sh`)
where `WL_HOME` is the directory in which you installed WebLogic Server.

Instead of using BEA's script, you can add the following JAR files to your environment's classpath, all of which are in the *WL_HOME*/server/lib/consoleapp/webapp/WEB-INF/lib directory:

- beehive-netui-tags-template.jar
- beehive-netui-tags-html.jar
- beehive-netui-scoping.jar
- console.jar
- controls.jar
- netuix_taglib.jar
- netuix_servlet.jar
- struts.jar

Import Tag Libraries Into IDEs (Optional)

If you are using BEA's JSP templates to create JSPs in your extension, you must use JSP tags from the JSP Standard Tag Library (JSTL), the BEA Administration Console Extension Tag Library, and the Apache Beehive Page Flows Tag Library.

The WebLogic Server runtime environment already provides these tag libraries. For development support, add the following tag libraries to your development environment:
WL_HOME/server/lib/consoleapp/webapp/WEB-INF/beehive-netui-tags-html.tld
WL_HOME/server/lib/consoleapp/webapp/WEB-INF/fmt.tld
WL_HOME/server/lib/consoleapp/webapp/WEB-INF/console-html.tld

Create a Directory Tree for the Extension

An Administration Console extension is a portion of a Web application and its resources must be organized into a directory structure that satisfies the requirements for standard J2EE Web applications. In addition, the WebLogic Portal framework, Apache Struts, and Apache Beehive require configuration files to be in specific locations.

To start working on your Administration Console extension, create a directory tree that matches the skeletal structure in Table 3-1.

Table 3-1 Directory Tree for an Administration Console Extension

Directory	Description
root-dir	The root directory of your extension. BEA recommends that you do not create files in this directory. The name of the directory has no programmatic significance. Choose a name that is meaningful to you. When specifying URIs in your extension, the "/" (forward slash) character by itself represents this root directory.
root-dir/WEB-INF	This directory must contain a file named `netuix-extension.xml`. This XML file functions as your extension's deployment descriptor. If you use Apache Struts, you must locate your Struts configuration file in this directory.
root-dir/WEB-INF/classes	If your extension uses a message bundle, your properties files must be in this directory. If your extension uses custom classes, your package structure must start in this directory. For example, if you packaged your class files in a package named `com.mycompany.extension`, then create the following directory structure in the `classes` directory: `com/mycompany/extension`. Then save your compiled class files in this `extension` directory.
(optional) *root-dir*/WEB-INF/src	If your extension uses custom classes, BEA recommends that you save your pre-compiled Java source files in a package structure that starts in this directory. When you archive your extension, you do not include this `src` directory.

Table 3-1 Directory Tree for an Administration Console Extension

Directory	Description
(recommended) *root-dir*/ext_jsp	BEA recommends that you save all of your extension's JSP files below a directory named ext_jsp. Creating a separate directory for your JSPs shields content developers from needing to learn about other support files such as the Portal framework XML files. If your extension contains many JSPs, consider creating subdirectories below ext_jsp. If you follow this recommendation, URIs for your JSPs will start with /ext_jsp. For example, /ext_jsp/myContent.jsp The directory named *root-dir*/jsp is reserved. **The root directory of your extension must not contain a directory named jsp**.
(recommended) *root-dir*/controls	BEA recommends that you save all of your extension's portal include files (.pinc) below a directory named controls. If your extension contains many books or pages, consider creating subdirectories below controls. If you follow this recommendation, URIs for your books or pages will start with /controls. For example, /controls/myBook.pinc
(recommended) *root-dir*/portlets	BEA recommends that you save all of your extension's portlet files (.portlet) below a directory named portlets. If your extension contains many portlets, consider creating subdirectories below portlets. If you follow this recommendation, URIs for your portlets will start with /portlets. For example, /portlets/myContent.portlet

If you are extending the Administration Console's Look and Feel, your root directory will contain additional subdirectories. See "Copy and Modify the Sample Look and Feel: Main Steps" on page 5-2.

Deploy a Development Look and Feel to See UI Control Labels

WebLogic Server provides a Look and Feel that reveals the labels of the Administration Console's extension points. You use these labels to specify where you want your extension to display.

Note: If you plan only to create a Look and Feel extension or add a portlet to the desktop, you do not need to deploy the development Look and Feel.

To use this Look and Feel:

1. Download the Look and Feel archive from the Code Samples section of the dev2dev Web site.

   The archive is distributed in a Code Samples project named Console Extension Developer Look and Feel (code-sample ID S118) and is available at the following URL:

   https://codesamples.projects.dev2dev.bea.com/servlets/Scarab?id=S118

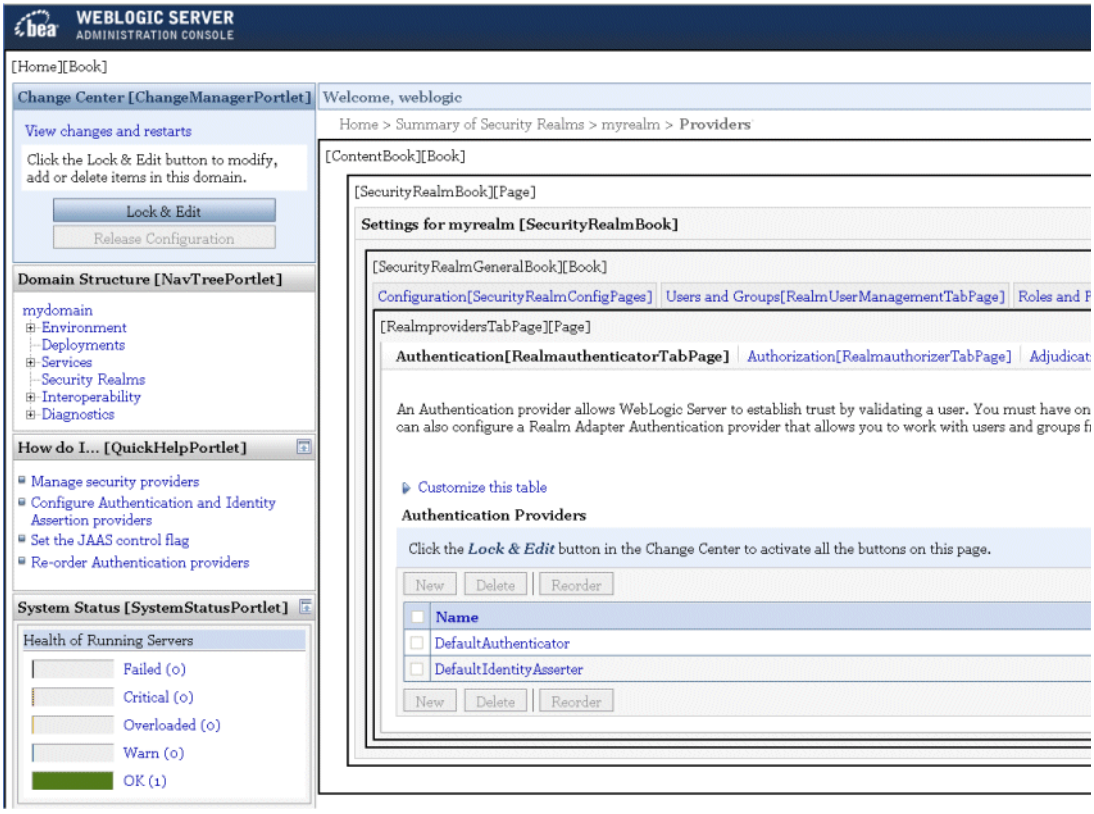
2. From the archive that you download, extract the `devlaf-1.0.jar` file and save it in *`domain-root`*`/console-ext`
   where *`domain-root`* is the root directory of any WebLogic Server domain in your development environment.

3. Restart the domain's Administration Server.

4. Log in to the Administration Console.

Each labeled control displays the value of its `definitionLabel` in brackets ([]) next to its user-visible title. In a separate pair of brackets, the control displays whether it is a book or a page. See Figure 3-1.

Figure 3-1 A Control Label in the Administration Console User Interface



Creating a Message Bundle

BEA recommends that you define all of the text strings that your Administration Console extension displays in a message bundle. A message bundle is a collection of text files (properties files) that contain key-value pairs (properties). You create one properties file for each language or locale that you want to support. If you name the properties file per a set of file-naming conventions, the Administration Console displays strings from the properties file whose locale matches the Web browser's locale setting.

Create a Message Bundle

To create a message bundle:

1. Create a text file that contains name-value pairs for each string you want to display. Use the equal sign (=) as the delimiter between the name and value, and place each property on its own line.

   For example:

   ```
   myextension.myTab.introduction=This page provides monitoring data for
   my application.
   myextension.myTab.TotalServletHits.label=Total hits for my servlet.
   ```

2. Save the file as *root-dir*/WEB-INF/classes/*bundle*.properties where
   - *root-dir* is the root directory of your extension
   - *bundle* is a unique value (do not use global, which is the name of a WebLogic Server bundle). Consider using your company name as the value for *bundle*.

   The *bundle*.properties file is the default file that the Administration Console uses if the Web browser or the JVM have not specified a locale. It is a required file.

3. Save each localized version of the properties file as *root-dir*/WEB-INF/classes/*bundle_locale*.properties

   where *locale* is a locale code supported by java.util.Locale. See Locale in the *J2SE API Specification*.

   For example, mycompany_ja.properties.

For information about using message bundles, see "Use a Message Bundle for Your Look and Feel" on page 5-7 and "Create and Use a Message Bundle in Your JSPs" on page 7-3.

CHAPTER 5

Rebranding the Administration Console

This section describes how to create a WebLogic Portal Look and Feel and deploy it as an Administration Console extension. The extension enables you to replace some or all of BEA's logos, colors, and styles in the Administration Console.

Figure 5-1 illustrates the process. The steps in the process, and the results of each are described in Table 5-1. Subsequent sections detail each step in the process.

Figure 5-1 Administration Console Extension Development Overview

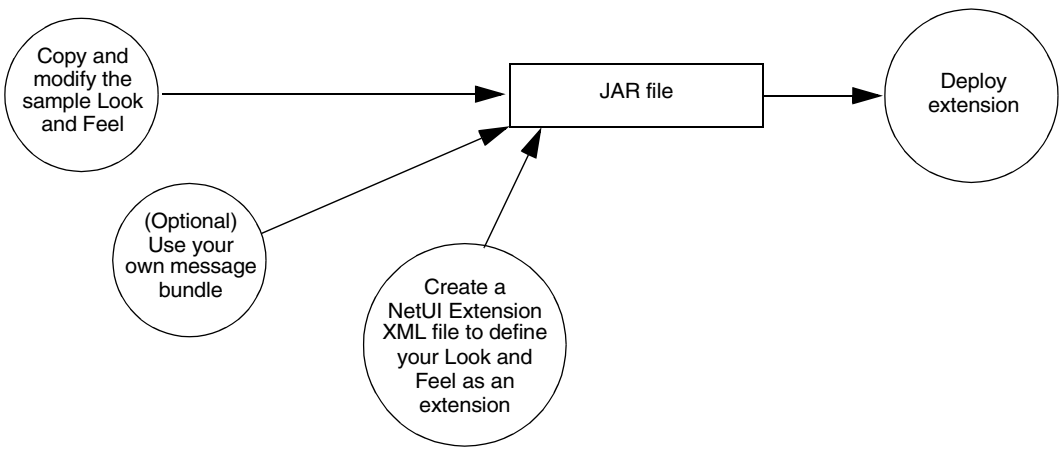


Table 5-1 Model MBean Development Tasks and Results

Step	Description	Result
1. "Copy and Modify the Sample Look and Feel: Main Steps" on page 5-2.	BEA installs a sample Look and Feel that you use as a starting point. Replace the images and styles in this sample with your own.	A Look and Feel that contains your logos and styles.
2. (Optional) "Use a Message Bundle for Your Look and Feel" on page 5-7.	If you want to change the text messages displayed in the banner, login, and login error pages, create your own message bundle and modify the pages to use messages from your bundle.	Localized properties files that contain your messages.
3."Modify the Sample NetUI Extension File" on page 5-9.	The NetUI Extension file is the deployment descriptor for your extension. It describes the locations of files and directories in your Look and Feel.	A deployment descriptor for your extension.
4. Archive and deploy the extension.	Archive the Look and Feel extension in a JAR file and copy it to your domain's `console-ext` directory. See "Archiving and Deploying Console Extensions" on page 8-1.	When the Administration Console starts in your domain, it uses the Look and Feel extension that is in the domain's `console-ext` directory instead of the Look and Feel that BEA packages and installs.

Copy and Modify the Sample Look and Feel: Main Steps

To create a simple extension that replaces the BEA logos and colors with your own:

1. Copy all files from the following directory into your own development directory:

   `WL_HOME/samples/server/xray/console-extension`

   where *`WL_HOME`* is the directory in which you installed WebLogic Server.

2. Change the name of the `xray` directory under *`root-dir`*`/framework/skins` and *`root-dir`*`/framework/skeletons` to a name that you choose.

   where *`root-dir`* is the name of your development directory.

   For example, *`root-dir`*`/framework/skins/mycompany` and *`root-dir`*`/framework/skeletons/mycompany`

3. "Modify the Administration Console Banner" on page 5-3.
4. "Modify Colors, Fonts, Buttons, and Images" on page 5-4.
5. "Modify Themes for the Change Center and Other Portlets" on page 5-5.
6. "Modify the Login and Error Page" on page 5-7.

Making more complex changes to the WebLogic Server Look and Feel, such as changing the layout of portal components and navigation menus, requires an advanced knowledge of WebLogic Portal Look and Feels. If you have a license for WebLogic Workshop, you can use its Look and Feel editor to make these complex changes. For more information, see the *Portal User Interface Framework Guide*.

Modify the Administration Console Banner

To overwrite the MedRec Look and Feel's image files with your company's image files:

1. To replace the logo in the Administration Console banner, save your own logo file as *`root-dir`*`/framework/skins/`*`mycompany`*`/images/banner_logo.gif`.

   To prevent the need to resize the banner frame, do not make your image any taller than 42 pixels.

2. To replace the ALT text for the logo, open *`root-dir`*`/framework/skeletons/`*`mycompany`*`/header.jsp` and replace `<bean:message key="login.wlsident">` with your text.

   If you want to provide localized strings, use the JSTL `<fmt:message>` tag. See "Use a Message Bundle for Your Look and Feel" on page 5-7.

3. To change the background color of the banner, replace the following image file with one of the same size but that contains a different color: *`root-dir`*`/framework/skins/`*`mycompany`*`/images/banner_bg.gif`

To make more complex modifications, you can change the JSP and styles that render the banner. The *`root-dir`*`/framework/skeletons/`*`mycompany`*`/header.jsp` file determines the contents of the Administration Console banner. Within `header.jsp`, the style

`bea-portal-body-header` specifies the name and location of an image file that is used as the banner background. The style `bea-portal-body-header-logo` specifies the name and location of the logo file. Both of these styles are defined in *`root-dir`*`/framework/skins/`*`mycompany`*`/css/body.css`.

Modify Colors, Fonts, Buttons, and Images

The Administration Console uses several cascading style sheets (CSS) to specify its fonts and colors. To change these styles, open the style sheet and change the style's definition. Table 5-2 summarizes the CSS files that the Administration Console uses. All of these files are located in the *`root-dir`*`/framework/skins/`*`mycompany`*`/css` directory.

Table 5-2 CSS Files That Define General Colors and Fonts

CSS File	Description
`wls.css`	Contains WebLogic Server styles for the following areas: • General definitions for elements such as `body`, `a`, `h1`, and `h2` • Data tables • Form fields • WebLogic Server form buttons • Error messages • Toolbar content • Breadcrumbs content • General styles for How Do I..., System Status, and Change Center portlets
• `body.css` • `book.css` • `button.css` • `form.css` • `layout.css` • `portlet.css` • `window.css`	Contain WebLogic Portal framework styles for the following areas (some of which are not used by the Administration Console): • Portal header and footer • Book, page, and menu styles • Button styles • Form, input, and text area styles • Layout and placeholder styles • Portlet styles

The buttons in the Administration Console use a repeating background image to render the blue fade (and grey for inactive buttons). The image files for these buttons are located in the following directory:

root-dir/framework/skins/*mycompany*/images

Modify Themes for the Change Center and Other Portlets

Several portlets in the Administration Console use a theme, and you can change the definitions of these themes. Themes are similar to Look and Feels but the scope of a theme is limited to a section of a portal, such as a book, page, or portlet. A theme can be used to change the look and feel of the components of a portal without affecting the portal itself.

For example, the Change Center portlet uses its own theme to distinguish its buttons from the other form buttons in the Administration Console.

To change the color of a theme's buttons or title bars, change the images and styles in the theme's `skins` directory. Table 5-2 summarizes the directories that contain CSS files and images for theme skins. All of these directories are under the *root-dir*/framework/skins/*mycompany* directory. For information about modifying skin themes, see Creating Skins and Skin Themes in *WebLogic Workshop Online Help*.

Table 5-3 Skins for Administration Console Themes

Skin Directory	Description
`wlsbreadcrumbs`	Defines fonts and spacing for the breadcrumbs portlet, which displays above the tabbed interface and provides a navigation history.
`wlschangemgmt`	Defines buttons, fonts, title bar background, and spacing for the Change Center portlet.
`wlsmessages`	Defines buttons, fonts, title bar background, and spacing for the messages portlet, which displays only when the Administration Console has validation or confirmation messages.
`wlsnavtree`	Defines buttons, fonts, title bar background, and spacing for the NavTreePortlet.
`wlsquicklinks`	Defines buttons, fonts, title bar background, and spacing for the How Do I... portlet.
`wlsstatus`	Defines buttons, fonts, title bar background, and spacing for the System Status portlet.
`wlstoolbar`	Defines fonts and spacing for the breadcrumbs portlet, which displays in the banner and contains the Home, Help, and AskBEA buttons.
`wlsworkspace`	Defines borders, spacing, and background colors of the books and pages in the ContentBook area of the Administration Console.

Each theme is made up of a skin **and** a skeleton. The skeleton defines the overall structure of the portlet contents. The definition for each theme's skeleton is under the `root-dir/framework/skeletons/mycompany` directory. For information about modifying skeleton themes, see Creating Skeletons and Skeleton Themes in *WebLogic Workshop Online Help*.

Modify the Login and Error Page

The login page asks users to enter a user ID and password. The login error page displays if users enter invalid data. Both of these pages are displayed before the Administration Console loads its portal desktop. Therefore, these pages do not use the portal's Look and Feel and their image and stylesheet files are not under the *root-dir*/framework directory. Table 5-4 summarizes the files and directories that determine the appearance of the login and login error pages.

Table 5-4 Files for the Login and Login Error Page Appearance

File	Description
root-dir/common/ login.css	Defines fonts and spacing for the login page.
root-dir/images/ login_banner_bg.gif login_banner_right.gif login_banner.gif login_bottom.gif	Images for the login page.
root-dir/login/ LoginError.jsp LoginForm.jsp	Render the login and login error pages. If you want to change the text that these pages display, modify the <fmt:message/> JSP tags to point to messages in your own message bundle. See "Use a Message Bundle for Your Look and Feel" on page 5-7.

Use a Message Bundle for Your Look and Feel

In the banner, login, and login error pages, the Administration Console uses JSTL tags to load text messages from localized properties files. For example, to display the window title in LoginForm.jsp:

1. The <fmt:setBundle basename="global" var="current_bundle" scope="page"/> tag in LoginForm.jsp sets the current message bundle to global.

   This JSP tag looks in WEB-INF/classes for files with the following name pattern: *bundle*[_*locale*].properties.

   The default properties file for this bundle is WEB-INF/classes/global.properties. If the Web browser or operating system specifies a different locale, then the JSP tag would load WEB-INF/classes/global_*locale*.properties.

2. The `<fmt:message key="window.title" bundle="${current_bundle}" />` tag opens the `global.properties` file and renders the text that is identified by the `window.title` key:
   `window.title=BEA WebLogic Server Administration Console`

If you want to change these messages, you can create your own properties files and modify the JSP tags to use your bundle. See "Creating a Message Bundle" on page 4-1.

Table 5-5 describes the text messages that the banner, login, and login error pages display.

Table 5-5 Messages in Banner, Login, and Login Error Pages

File	Message Key and Value
root-dir/login/LoginForm.jsp	window.title=BEA WebLogic Server Administration Console login.wlsident=BEA WebLogic Server Administration Console login.welcome2=Log in to work with the WebLogic Server domain login.username=Username: login.password=Password: login.submit=Log In
root-dir/login/LoginError.jsp	window.title=BEA WebLogic Server Administration Console login.wlsident=BEA WebLogic Server Administration Console loginerror.authdenied=Authentication Denied loginerror.passwordrefused=The username or password has been refused by WebLogic Server. Please try again. login.username=Username: login.password=Password: login.submit=Log In
root-dir/framework/skeleton/*mycompany*/header.jsp	window.title=BEA WebLogic Server Administration Console

Modify the Sample NetUI Extension File

A NetUI Extension file is the deployment descriptor for your Look and Feel. It contains the names and locations of the files in your Look and Feel, and it causes the Administration Console to replace its Look and Feel with yours. For more information, see the *NetUI Extensions Schema Reference*.

The sample file is in the following location:
`root-dir/WEB-INF/netuix-extension.xml`

To modify this file:

1. Open the file in a validating XML editor (recommended) or a text editor.
2. In the `<provider-info>` element, change the information to describe your Look and Feel, developer contact and support URL.

   The information in this element has no programmatic significance. It is intended to help your technical support team keep track of your software modifications.
3. In the `<look-and-feel-content>` element:

   a. In the `title`, `skin`, and `skeleton` attributes, replace the `medrec` value with the name of the directory you chose in step 2 in "Copy and Modify the Sample Look and Feel: Main Steps" on page 5-2.

   b. In the `definitionLabel` and `markupName` attributes, replace the `medrec` value with the name of the directory you chose in step 2 or use some other string. These attributes are required by the portal framework, but are not used in a Look and Feel extension.

CHAPTER 6

Adding Portlets and Navigation Controls

In the Administration Console, all content is contained within portlets, so even the most minimal extension must define a portlet (and content for the portlet). You can add your portlet directly to the desktop, but if you want the portlet to display as a tab or subtab in the `ContentBook`, you must define books or pages to contain it. Your extension can also add a node to the NavTreePortlet, which enables users to navigate to your portlet directly from the desktop.

This section describes how to add portlets, UI controls, and NavTreePortlet nodes to the Administration Console.

Figure 6-1 illustrates the process. The steps in the process, and the results of each are described in Table 6-1. Subsequent sections detail each step in the process.

Figure 6-1 Adding Portlets and Navigation Controls Development Overview

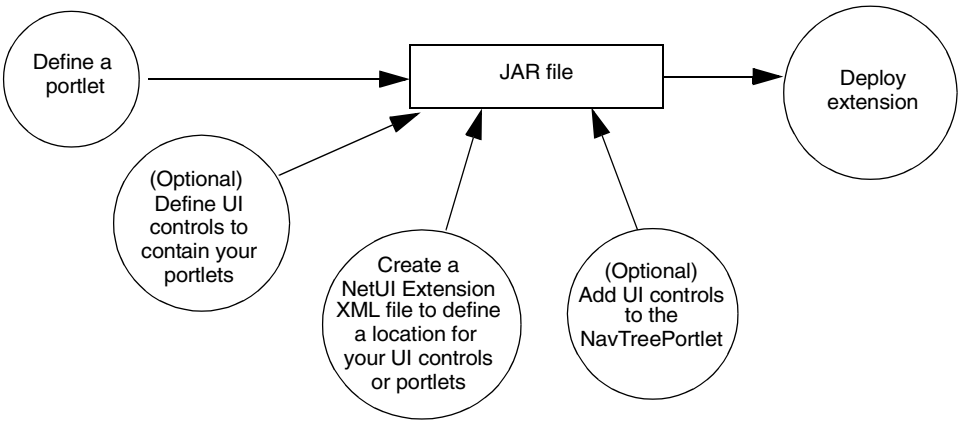


Table 6-1 Model MBean Development Tasks and Results

Step	Description	Result
1. "Define a Portlet" on page 6-3.	Create an XML file to define a portlet that the portal framework can instantiate. A portlet definition includes instructions on which type of data to load: JSPs, Struts Actions, or Beehive Page Flows. The portal's Look and Feel determines whether the portlet provides borders and minimize/maximize controls.	A `.portlet` XML file.
2."Define UI Controls (Optional)" on page 6-12.	If you want your portlet to display in a tab, subtab, or in some other location within `ContentBook`, create an XML file that defines a page or book.	A `.pinc` XML file.
3. "Specify a Location for Displaying Portlets or UI Controls" on page 6-18.	Create an XML file that describes whether you want your portal to display next to a labeled UI control or to replace the control.	A `netuix-extension.xml` file.
4. "Add Nodes to the NavTreePortlet (Optional)" on page 6-22.	You can create a link from the NavTreePortlet to any book or page in your extension. WebLogic Server provides default support for appending control names to the end of the existing navigation tree. If you want to insert nodes in specific locations, or if you want to create a node tree, you create your own Java classes that describe the node and node location.	Additional entries in the `.pinc` XML file. Optionally, Java classes that give you more control over the node that you are adding.
5. Archive and deploy the extension.	See "Archiving and Deploying Console Extensions" on page 8-1.	A JAR file that contains your extension.

Define a Portlet

You define a portlet in an XML file. The portlet definition includes instructions on which type of data to load: JSPs, Struts Actions, or Beehive Page Flows. The following sections describe how to define a portlet:

- "Define a JSP Portlet" on page 6-3
- "Define a Struts Portlet" on page 6-4
- "Define a Page Flow Portlet" on page 6-6
- "Displaying a Title Bar for a Portlet" on page 6-7

For more information about portlet XML files, see the portlet entry in *Portal Support Schema Reference*.

Define a JSP Portlet

To define a portlet that loads a JSP:

1. Copy the code from Listing 6-1 and paste it into a new text file in *root-dir*/portlets (see "Create a Directory Tree for the Extension" on page 3-2).

   Consider using the following naming convention:
   content-name.portlet

   where *content-name* is the name of a JSP file that the portlet contains. For example, if the portlet contains a JSP file named monitorEJB.jsp, then name the portlet XML file monitorEJB.portlet.

2. Replace the values in Listing 6-1 as follows:
   - *Label*. Provide a unique identifier that the portal framework uses to identify this portlet.
   - (optional) *Title*. Provide a default title that this portlet displays if its title bar is visible. See "Displaying a Title Bar for a Portlet" on page 6-7.
   - *URI*. Specifies the absolute path and file name of the JSP that the portlet contains starting from the root of the extension.

     For example:
     /ext_jsp/monitorEJB.JSP

Listing 6-1 Template for a Portlet XML File that Loads a JSP File

```
<?xml version="1.0" encoding="UTF-8"?>
<portal:root xmlns:html="http://www.w3.org/1999/xhtml-netuix-modified/1.0.0"
   xmlns:portal="http://www.bea.com/servers/netuix/xsd/portal/support/1.0.0"
   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/portal/
      support/1.0.0 portal-support-1_0_0.xsd">

   <netuix:portlet definitionLabel="Label" title="Title" >
      <netuix:content>
         <netuix:jspContent contentUri="URI"/>
      </netuix:content>
   </netuix:portlet>

</portal:root>
```

Define a Struts Portlet

Instead of encapsulating your extension's business logic and navigation logic in JSP files, you can use the Apache Struts framework. See "Create Struts Artifacts for Tables and Forms" on page 7-7.

To create a portlet that loads (forwards to) a Struts Action:

1. Copy the code from Listing 6-2 and paste it into a new text file in *root-dir*/portlets (see "Create a Directory Tree for the Extension" on page 3-2).

   Consider using the following naming convention:
   action-name.portlet

   where *action-name* is the name of the Struts Action to which the portlet forwards.

2. Replace the values in Listing 6-2 as follows:
   - *Label*. Provide a unique identifier that the portal framework uses to identify this portlet.
   - (optional) *Title*. Provide a default title that this portlet displays if its title bar is visible. See "Displaying a Title Bar for a Portlet" on page 6-7.
   - *Struts-module*. Specifies the Struts module that defines a Struts Action.

     You must create your own Struts module to define the Actions and ActionForms that your Administration Console extension uses; the default Struts module is reserved for BEA Actions and ActionForms. Each module includes its own, uniquely named configuration file. For information about Struts modules, see the Apache Struts *User Guide* at http://struts.apache.org/struts-doc-1.2.x/userGuide/index.html.

     For example, if you specify "myModule" for *Struts-module*, the Struts controller servlet looks in the following location for the action:
     root-dir/WEB-INF/struts-auto-config-myModule.xml
   - *action-path*. Specifies the path to a Struts Action that is defined in your Struts module.
   - *refresh-action-path*. Specifies the Action to invoke on subsequent requests for this portlet (for example, the user agent refreshes the document).

   Note that this .portlet does not specify the name of a JSP. Instead, typically the Struts Action mapping forwards to a specific JSP upon successful operation.

Listing 6-2 Template for a Portlet XML File that Forwards to a Struts Action

```
<?xml version="1.0" encoding="UTF-8"?>
<portal:root xmlns:html="http://www.w3.org/1999/xhtml-netuix-modified/1.0.0"
   xmlns:portal="http://www.bea.com/servers/netuix/xsd/portal/support/1.0.0"
   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/portal/
      support/1.0.0 portal-support-1_0_0.xsd">

   <netuix:portlet definitionLabel="Label" title="Title" >
     <netuix:strutsContent module="Struts-module"
          action="action-path"
          refreshAction="refresh-action-path"/>
   </netuix:portlet>
</portal:root>
```

Define a Page Flow Portlet

To define a portlet that loads a Beehive Page Flow:

1. Copy the code from Listing 6-3 and paste it into a new text file in *root-dir*/portlets (see "Create a Directory Tree for the Extension" on page 3-2).

   Consider using the following naming convention:
   pageFlow-name.portlet

   where *pageFlow-name* is the name of the Page Flow that the portlet loads (forwards to). For example, if the portlet forwards to a Page Flow named myPageFlow.jpf, then name the portlet XML file myPageFlow.portlet.

2. Replace the values in Listing 6-3 as follows:

   – *Label*. Provide a unique identifier that the portal framework uses to identify this portlet.

   – (optional) *Title*. Provide a default title that this portlet displays if its title bar is visible. See "Displaying a Title Bar for a Portlet" on page 6-7.

   – *URI*. Specifies the absolute path and file name of the JPF file that defines the Page Flow. The URI must be absolute starting from the *root-dir*/WEB-INF/classes directory.

     For example, if your JPF file is *root-dir*/WEB-INF/classes/com/mycompany/extension/pageflows/myPageFlow.jpf, specify the following value
     /com/mycompany/extension/pageflows/myPageFlow.jpf

   – *Action*. Specifies the absolute path and file name of the JPF file that defines the Page Flow.

Listing 6-3 Template for a Portlet XML File that Forwards to a Page Flow

```
<?xml version="1.0" encoding="UTF-8"?>
<portal:root xmlns:html="http://www.w3.org/1999/xhtml-netuix-modified/1.0.0"
   xmlns:portal="http://www.bea.com/servers/netuix/xsd/portal/support/1.0.0"
   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/portal/
      support/1.0.0 portal-support-1_0_0.xsd">

   <netuix:portlet definitionLabel="Label" title="Title" >
      <netuix:content>
          <netuix:pageflowContent
                 contentUri="URI"
                 action="Action"/>
                 refreshAction="refresh-Action"/>
      </netuix:content>
   </netuix:portlet>

</portal:root>
```

Displaying a Title Bar for a Portlet

If you plan to locate a portlet on the Administration Console desktop (within a placeholder on the "page" page), configure the portlet to display a title bar. If you locate a portlet in the ContentBook, do not display a title bar.

To display a title bar:

1. In the portlet's .portlet XML file, provide a value for the title attribute of the netuix:portlet element. To display a localized value, see "Localizing a Portlet Title" on page 6-8.
2. Include the following element as a child of the netuix:portlet element:

   ```
   <netuix:titlebar/>
   ```

   To enable the portlet to be minimized and maximized, include the following stanza instead of the empty <netuix:titlebar/> element:

   ```
   <netuix:titlebar>
      <netuix:minimize/>
      <netuix:maximize/>
   </netuix:titlebar>
   ```

Listing 6-4 defines a portlet that displays a title bar. The portlet can be minimized or maximized and the title value comes from a message bundle.

Listing 6-4 Example: Portlet that Displays a Localized Title

```
<?xml version="1.0" encoding="UTF-8"?>
<portal:root xmlns:html="http://www.w3.org/1999/xhtml-netuix-modified/1.0.0"
   xmlns:portal="http://www.bea.com/servers/netuix/xsd/portal/support/1.0.0"
   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/portal/
      support/1.0.0 portal-support-1_0_0.xsd">

   <netuix:portlet definitionLabel="medrecEAR.Monitor.Portlet"
       title="medrecMBean.myPortlet.title"
       backingFile="com.bea.medrec.extension.utils.DesktopViewBacking">
      <netuix:titlebar>
          <netuix:minimize/>
          <netuix:maximize/>
      </netuix:titlebar>
      <netuix:content>
         <netuix:strutsContent module="/medrecMBean"
              action="RetrieveCustomMBeansAction"
               refreshAction="RetrieveCustomMBeansAction"/>
      </netuix:content>
   </netuix:portlet>
</portal:root>
```

Localizing a Portlet Title

By default, the portlet displays the literal value that you enter in the `<netuix:portlet>` element's `title` attribute. To enable this title to be localized:

1. Create a Java class that retrieves the value of the `title` attribute, scans a property file for a key that matches the `title` attribute value, and returns the value of the property key.

   For example, if you specify `title="myPortlet.title"`, the Java class looks through your message bundle for `myPortlet.title=MyCompany's Portlet` and returns `MyCompany's Portlet` as the text to be displayed.

   See "Create a Backing Class for Localizing Portlet Titles" on page 6-9.

2. In the `.portlet` file, include the following attributes in the `<netuix:portlet>` element:
   - `title`. Specify the key for a property that you have defined in your message bundle.
   - `backingFile`. Specify the fully-qualified name of a Java class that you created in the previous step.

   For example:
   ```
   <netuix:portlet definitionLabel="myPortlet" title="myPortlet.title"
         backingFile="com.mycompany.extension.utils.MyPortletBacking">
   ```

Create a Backing Class for Localizing Portlet Titles

A backing class is a Java class that interacts directly with the portal framework APIs. To create a backing class that retrieves localized portlet titles:

1. Extend `com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking`.
2. Implement the `AbstractJspBacking.preRender(HttpServletRequest request, HttpServletResponse response)` method.

   See `AbstractJspBacking.preRender()` in the *WebLogic Portal API Reference*.

   In your implementation of this method:

   a. Get the locale from the `HttpServletRequest` object.

      Use the following API:
      ```
      javax.servlet.http.HttpServletRequest.getSession().getAttribute(
      "org.apache.struts.action.LOCALE")
      ```

   b. Get the message bundle.

      Use the following API:
      ```
      org.apache.struts.util.MessageResources.getMessageResources(
          "myBundle");
      ```
      where *`myBundle`* is the name of your message bundle. (See "Creating a Message Bundle" on page 4-1.)

c. Get the value of the portlet's `title` property.

   Use the following APIs:

```
PortletBackingContext bctx =
     PortletBackingContext.getPortletBackingContext(
         HttpServletRequest req);
MessageResources.getMessage(locale, bctx.getTitle());
```

   where *`locale`* is the locale that you retrieved from the `HttpServletRequest` object.

d. Reset the value of the portlet's `title` property to the localized value that you retrieved in the previous step.

   Use the following API:

   `PortletBackingContext.getTitle(String` *`title`*`)`

   where *`title`* is the value that you retrieved from the message bundle.

Listing 6-5 Example: Backing Class for Localizing a Portlet Title

```
package com.bea.medrec.extension.utils;

import java.util.Locale;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.util.MessageResources;

import com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking;
import com.bea.netuix.servlets.controls.portlet.backing.PortletBackingContext;

public class DesktopViewBacking extends AbstractJspBacking {
   public boolean preRender(HttpServletRequest req, HttpServletResponse res) {
      // Get the PortletBackingContext for current portlet. The
      // PortletBackingContext contains properties and methods
      // for the current portlet.
      PortletBackingContext bctx =
         PortletBackingContext.getPortletBackingContext(req);

      if (bctx != null) {
         // If title does not contain a period, assume it's preLocalized
         // or follow the format for a key
         if (bctx.getTitle().indexOf(".")!=-1){
            // Get the locale from the HTTPServletRequest
            Locale locale = (Locale) req.getSession().getAttribute(
               "org.apache.struts.action.LOCALE");
            // Find the message bundle named "medrecMBean"
            MessageResources messages =
               MessageResources.getMessageResources("medrecMBean");
            // Get the value of the portlet's "title" property
            String msg = messages.getMessage(locale, bctx.getTitle());
            // Reset the value of the "title" property with the
            // localized value.
            bctx.setTitle(msg);
         }
      }
      return true;
   }
}
```

Define UI Controls (Optional)

If you want to add tabs or subtabs to the Administration Console, you must define a book or page UI control that conforms to the existing hierarchy:

- To create a top-level tab (such as a sibling of Domains: Configuration), you create a book that contains one or more pages. Each page contains a portlet.
- To create a subtab of an existing tab (such as a sibling of Domains: Configuration: General), you create a page that contains a portlet.

Save the definitions of your books and pages in one or more portal include (`.pinc`) files. Create one `.pinc` file for each hierarchical grouping of controls. For example, create one `.pinc` file for a book that creates a top-level tab and its subtabs. Create another `.pinc` file for a page that adds a subtab to an existing WebLogic Server tab. The root element of a `.pinc` file (`portal:root`) can have only one direct child element; the child element can have multiple children.

The following sections describe creating books and pages:

- "Create a Tab That Does Not Contain a Subtab" on page 6-12
- "Create a Tab That Contains Subtabs" on page 6-14
- "Create a Subtab" on page 6-18
- "Create a Control Without Tabs or Subtabs" on page 6-18

Create a Tab That Does Not Contain a Subtab

To create a portal include (`.pinc`) XML file that defines a tab and no subtabs (such as Domains: Notes):

1. Copy the code from Listing 6-6 and paste it into a new text file.

   For example, *`root-dir`*`/controls/MyApp.pinc`
   where *`root-dir`* is your development directory. For more information, see "Setting Up a Development Environment" on page 3-1.

2. Replace the values in Listing 6-6 as follows:

   – *Page-Label*. Provide a unique identifier that the portal framework and WebLogic Server JSP tags use to forward requests to the page.

   – *Page-Title*. Provide either the text that users see as the name of the tab or a key in a message bundle that you have created.

     If the value that you specify contains a "." (period), the Administration Console assumes that this value is a key and attempts to look up the value from your message bundle. For example, if you specify `My.Tab`, the Administration Console looks up the value of a property whose key is `My.Tab`. If it cannot find such a value, it displays `null` as the tab name. If you specify `My Tab` as the value, then the Administration Console displays `My Tab`.

   – *Bundle*. Specify the name of a message bundle that you have created. This bundle is used only if the value of the `title` attribute in the `netuix:page` element contains a ".". See "Create and Use a Message Bundle in Your JSPs" on page 7-3.

   – *Portlet-Instance-Label*. Provide a unique identifier that the portal framework and WebLogic Server JSP tags use to forward requests to the portlet instance.

   – *Portlet-URI*. Specify the path and file name of a portlet file that you created (see "Define a Portlet" on page 6-3). The path must be relative to the root of the portal Web application.

     For example:

     ```
     /portlets/monitorEJB.portlet
     ```

Note that Listing 6-6 defines a page, not a book, so the Administration Console Look and Feel will render the page as a tab with no subtabs.

Listing 6-6 Template .pinc File that Creates a Tab with No Subtabs

```
<?xml version="1.0" encoding="UTF-8"?>
<portal:root
   xmlns:netuix="http://www.bea.com/servers/netuix/xsd/controls/netuix/1.0.0"
   xmlns:html="http://www.w3.org/1999/xhtml-netuix-modified/1.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:portal="http://www.bea.com/servers/netuix/xsd/portal/support/1.0.0"
   xsi:schemaLocation="http://www.bea.com/servers/netuix/xsd/portal/support
      /1.0.0 portal-support-1_0_0.xsd">
   <netuix:page markupName="page" markupType="Page"
      definitionLabel="Page-Label" title="Page-Title"
      skeletonUri="/framework/skeletons/default/wlsworkspace/
         page_content.jsp">
         <netuix:meta name="skeleton-resource-bundle" content="Bundle"/>
         <netuix:content>
            <netuix:gridLayout columns="1" markupType="Layout"
               markupName="singleColumnLayout">
            <netuix:placeholder flow="vertical" markupType="Placeholder"
               markupName="singleColumn_columnOne">
               <netuix:portletInstance markupType="Portlet"
                  instanceLabel="Portlet-Instance-Label"
                  contentUri="Portlet-URI"/>
            </netuix:placeholder>
         </netuix:gridLayout>
      </netuix:content>
   </netuix:page>
</portal:root>
```

Create a Tab That Contains Subtabs

To create a portal include (`.pinc`) XML file that defines a tab and one or more subtabs:

1. Copy the code from Listing 6-7 and paste it into a new text file. Save the file in a directory below *`root-dir`*.

   For example, *`root-dir`*`/controls//MyApp.pinc`
   where *`root-dir`* is your development directory. For more information, see "Setting Up a Development Environment" on page 3-1.

2. To define the tab, replace the values in Listing 6-7 as follows:
   - *Book-Label*. Provide a unique identifier that the portal framework and WebLogic Server JSP tags use to forward requests to the book. This is the same type of label that WebLogic Server provides for many of its UI controls. See "Extension Points in the Administration Console" on page 2-3.
   - *Book-Title*. Provide either the text that users see as the name of the tab or a key in a message bundle that you have created.

     If the value that you specify contains a "." (period), the Administration Console assumes that this value is a key and attempts to look up the value from your message bundle. For example, if you specify `My.Tab`, the Administration Console looks up the value of a property whose key is `My.Tab`. If it cannot find such a value, it displays `null` as the tab name. If you specify `My Tab` as the value, then the Administration Console displays `My Tab`.
   - *Bundle*. Specify the name of a message bundle that you have created. This bundle is used only if the value of the `title` attribute in the `netuix:book` element contains a ".". See "Create and Use a Message Bundle in Your JSPs" on page 7-3.
3. To define the first subtab, replace the values in Listing 6-7 as follows:
   - *Page-Label*. Provide a unique identifier that the portal framework and WebLogic Server JSP tags use to forward requests to the page.
   - *Page-Title*. Provide either the text that users see as the name of the subtab or a key in a message bundle that you have created.

     If the value that you specify contains a "." (period), the Administration Console assumes that this value is a key and attempts to look up the value from your message bundle.
   - (optional) *Metadata-Type* and *Metadata-ID*. If you want to use the Administration Console's `<wl:column-dispatch>` JSP tag to create a hypertext link that forwards to this page, include a `<netuix:meta>` element and supply values for *Metadata-Type* and *Metadata-ID*. See "Create a Table Column for Navigating to Other Pages" on page 7-26.
   - *Portlet-Instance-Label*. Provide a unique identifier that the portal framework and WebLogic Server JSP tags use to forward requests to the portlet instance.
   - *Portlet-URI*. Specify the path and file name of a portlet file that you created (see "Define a Portlet" on page 6-3). The path must be relative to the root of the portal Web application.

For example:
`/portlets/monitorEJB.portlet`

4. To create additional subtabs, add `netuix:page` elements as siblings to the `netuix:page` element in Listing 6-7.

For more information about portal include XML files, see the *Portal Support Schema Reference*.

Note the use of the following elements in the `.pinc` file:

- `netuix:singleLevelMenu` renders one subtab for each page in the book. The book's parent UI control (which Listing 6-7 assumes is provided by WebLogic Server) is responsible for generating a top-level tab for the book.
- `netuix:meta name="breadcrumb-context" content="handle"` adds the page's title to the history of visited pages (breadcrumbs) after a user has visited the page. The breadcrumbs display on the desktop above `ContentBook`.

Listing 6-7 Template for a .pinc File That Defines a Top-Level Tab with Subtabs

```
<?xml version="1.0" encoding="UTF-8"?>
<portal:root
   xmlns:netuix="http://www.bea.com/servers/netuix/xsd/controls/netuix/1.0.0"
   xmlns:html="http://www.w3.org/1999/xhtml-netuix-modified/1.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:portal="http://www.bea.com/servers/netuix/xsd/portal/support/1.0.0"
   xsi:schemaLocation="http://www.bea.com/servers/netuix/xsd/portal/support
      /1.0.0 portal-support-1_0_0.xsd">

   <netuix:book markupName="book" markupType="Book"
      definitionLabel="Book-Label" title="Book-Title">
      <netuix:singleLevelMenu markupType="Menu" markupName="singleLevelMenu"
         skeletonUri="singlelevelmenu_children.jsp"/>
      <netuix:meta name="skeleton-resource-bundle" content="Bundle"/>
      <netuix:content>
         <netuix:page markupName="page" markupType="Page"
            definitionLabel="Page-Label" title="Page-Title"
            skeletonUri="/framework/skeletons/default/wlsworkspace/
               page_content.jsp">
            <netuix:meta name="Metadata-Type" content="Metadata-ID"/>
            <netuix:meta name="breadcrumb-context" content="handle"/>
            <netuix:meta name="skeleton-resource-bundle" content="Bundle"/>
            <netuix:content>
               <netuix:gridLayout columns="1" markupType="Layout"
                  markupName="singleColumnLayout">
                  <netuix:placeholder flow="vertical" markupType="Placeholder"
                     markupName="singleColumn_columnOne">
                     <netuix:portletInstance markupType="Portlet"
                        instanceLabel="Portlet-Instance-Label"
                        contentUri="Portlet-URI"/>
                  </netuix:placeholder>
               </netuix:gridLayout>
            </netuix:content>
         </netuix:page>
         <!-- Add additional netuix:page elements here -->
      </netuix:content>
   </netuix:book>
</portal:root>
```

Create a Subtab

To create a subtab that you can add to an existing WebLogic Server tab:

1. Create a `.pinc` file that defines a page UI control. See Listing 6-6.
2. In your `netuix-extension.xml` file, specify the WebLogic Server book UI control that you want to contain your subtab. See "Add a Tab or Subtab to ContentBook" on page 6-20.

Create a Control Without Tabs or Subtabs

There is no requirement for books and pages in `ContentBook` to be accessible by tab or subtab. Many WebLogic Server pages that display summary tables are accessible from the NavTreePortlet but not from the tabbed interface (see Figure 2-5).

Any of the code listings in the previous sections can be located in a parent control that does not render tabs or subtabs for its children. See "Specify a Location for Displaying Portlets or UI Controls" on page 6-18.

Specify a Location for Displaying Portlets or UI Controls

All locations for displaying your portlets or UI controls must be specified as relative to existing controls in the Administration Console. For example, you can specify that your portlet displays on the desktop below the System Status portlet.

To specify a location for displaying a portlet or UI control:

1. Create an XML file named `netuix-extension.xml` and save it in *`root-dir`*`/WEB-INF`
   where *`root-dir`* is your development directory. For more information, see "Setting Up a Development Environment" on page 3-1.

   A NetUI Extension XML file (`netuix-extension.xml`) is the deployment descriptor for your extension. It declares each parent UI control in your extension and the location in which you want it to display (see Listing 6-8). For more information, see the *NetUI Extensions Schema Reference*.
2. Create a `<weblogic-portal-extension>` root element.
3. (Optional) Create a `<provider-info>` element to describe your extension.

   This element is for your information only. The portal framework does not use the data in this element.

4. Add the following element:
   `<portal-file>/console.portal</portal-file>`

   This required element specifies the name and relative location of the Administration Console's `.portal` file, which is the portal that you are extending.

5. Do one of the following:
   - "Add a Portlet to the Desktop" on page 6-19
   - "Add a Tab or Subtab to ContentBook" on page 6-20

Add a Portlet to the Desktop

To add a portlet to the Administration Console desktop, create the following stanza in your `netuix-extension.xml` file (see Listing 6-8):

```
<page-extension>
   <page-location>
      <parent-label-location label="page"/>
      <page-insertion-point layout-location="layout"
         placeholder-position="0"/>
   </page-location>
   <portlet-content
      content-uri="portlet-URI" title="title"
      orientation="top" default-minimized="false"
      instance-label="portlet-instance-label"/>
</page-extension>
```

where:

- ***layout*** is one of the following values:
  - `0` (zero) if you want the portlet to display in the left side of the Administration Console.

    Extension portlets always display at the top of the left column.
  - `1` (one) if you want the portlet to display in the right side.

    Extension portlets always display at the bottom of the right column.
- ***portlet-URI*** is the path and file name of your `.portlet` file. The path must be relative to the root of the portal Web application.
- ***title*** is the title that displays in the portlet's title bar. If you specify a null value, the portal framework uses the title that you defined in the `.portlet` file.

- ***portlet-instance-label*** is a unique identifier that the portal framework and WebLogic Server JSP tags use to forward requests to the portlet instance.

Add a Tab or Subtab to ContentBook

To add a control that renders a tab, create the following stanza in your `netuix-extension.xml` file (see Listing 6-8):

```
<book-extension>
   <book-location>
      <parent-label-location label="Admin-Console-Book-Label"/>
      <book-insertion-point action="append"/>
   </book-location>
   <book-content content-uri="pinc-URI"/>
</book-extension>
```

where:

- ***Admin-Console-Book-Label*** is the `definitionLabel` of an Administration Console book control that renders tabs for its child books.
- ***pinc-URI*** is the path and file name of your `.pinc` file that defines the book control for your tab (and optional subtabs). The path must be relative to the root of the portal Web application.

To add a control that renders a subtab in an existing tab, create the same stanza as the previous step, where:

- ***Admin-Console-Book-Label*** is the `definitionLabel` of an Administration Console book control that renders subtabs for its child pages.
- ***pinc-URI*** is the path and file name of your `.pinc` file that defines the page control for your subtab. The path must be relative to the root of the portal Web application.

Example: Specifying Locations for Portlets and UI Controls

Listing 6-8 is a `netuix-extension.xml` file that adds a portlet to the console desktop, a tab to the WebLogic Sever Domain tabs, and subtab to the Domain: Configuration tab.

Listing 6-8 Example netuix-extension.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-portal-extension
   xmlns="http://www.bea.com/servers/portal/weblogic-portal/8.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/servers/portal/weblogic-portal/
      8.0 netuix-extension-1_0_0.xsd">
   <provider-info>
      <title>My Extension</title>
      <version>1.0</version>
      <description>Inserts a portlet on the desktop, a tab next to
           Domains:Configuration, and a subtab under Domains: Configuration.
      </description>
      <author>Me</author>
      <last-modified>02/03/2005</last-modified>
      <support-url>http://www.mycompany/support/index.jsp</support-url>
   </provider-info>

   <portal-file>/console.portal</portal-file>

   <!--Adds a portlet to the console desktop -->
   <page-extension>
      <page-location>
         <parent-label-location label="page"/>
         <page-insertion-point layout-location="0" placeholder-position="0"/>
      </page-location>
      <portlet-content content-uri="/portlets/desktop/desktop_view.portlet"
         title="My App Status" orientation="top" default-minimized="false"
          instance-label="PortletExtensionInstanceLabel"
      />
   </page-extension>

   <!--Adds a tab to the Domain tabs -->
   <book-extension>
      <book-location>
         <parent-label-location label="CoreDomainConfigGeneralBook"/>
         <book-insertion-point action="append"/>
      </book-location>
      <book-content content-uri="/controls/page.pinc"/>
   </book-extension>
```

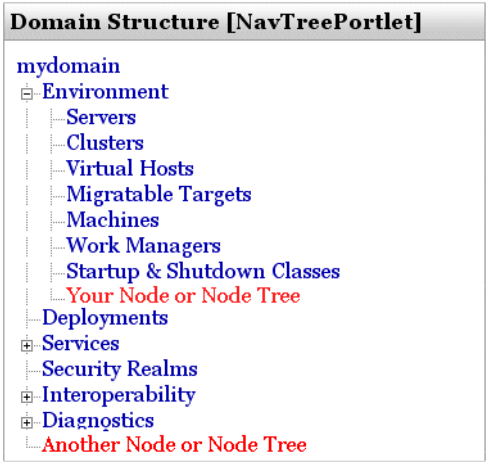
```
<!-- Adds a subtab to the Domain: Configuration tab-->
   <book-extension>
      <book-location>
         <parent-label-location label="DomainconfigTabPage"/>
         <book-insertion-point action="append"/>
      </book-location>
      <page-content content-uri="/controls/notespage.pinc"/>
   </book-extension>

</weblogic-portal-extension>
```

Add Nodes to the NavTreePortlet (Optional)

The Domain Structure portlet (NavTreePortlet) contains a tree control that you can use to navigate to content in the Administration Console. Each node in the tree is a link to a UI page control. Nodes can also contain subnodes.

Your extension can add a single node at any location in the tree. It can also add a node that contains other nodes (node tree) at any location. For example, your extension can add a node or a node tree to the root of the existing navigation tree. In addition (or instead), it can add a node or node tree to the Environments node. (See Figure 6-2.)

Figure 6-2 Example: Adding Nodes or Node Trees



The following sections describe adding nodes to the NavTreePortlet:

- "Append a Single Node to the Root of the Existing Tree" on page 6-23
- "Append or Insert Nodes or Node Trees" on page 6-24

Append a Single Node to the Root of the Existing Tree

To append a node that links to one of your page controls, add the following attribute and attribute value to the `netuix:page` element in the control's `.pinc` file:

```
backingFile="com.bea.console.utils.NavTreeExtensionBacking"
```

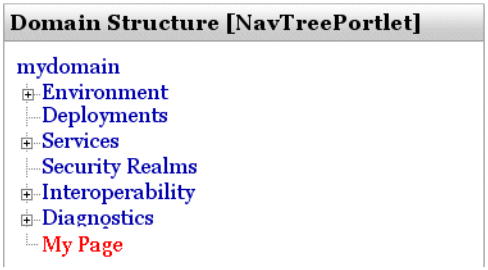
For example, if you want to add a link to a page that you have created, in the `.pinc` file that defines your page, add the `backingFile` attribute:

```
<netuix:page definitionLabel="MyAppTableBook" title="My Page"
   markupName="page"
   markupType="Page"
   backingFile="com.bea.console.utils.NavTreeExtensionBacking"
>
```

The NavTreePortlet displays the value of the page element's `title` attribute as the link text. See Figure 6-3.

If the `title` attribute value is a key in your message bundle, the NavTreePortlet displays the localized value mapped to the key.

Figure 6-3 Append a Node to the Root of the Existing Tree



Append or Insert Nodes or Node Trees

If you want to control the location in which your node is added to the NavTreePortlet, or if you want to add a node tree, implement your own `NavTreeExtensionBacking` backing class.

The following sections describe appending or inserting nodes or node trees:

Create a NavTreeBacking Class

To create a `NavTreeBacking` class (see Listing 6-9):

1. Extend `com.bea.console.utils.NavTreeExtensionBacking`.

   This class is already available in the WebLogic Server runtime environment. However, for support in your development and compiling environment, you must add the following JARs to your environment's classpath:
   `WL_HOME``/server/lib/consoleapp/webapp/WEB-INF/lib/console.jar`
   `WL_HOME``/server/lib/consoleapp/webapp/WEB-INF/lib/netuix_servlet.jar`

   where *`WL_HOME`* is the location in which you installed WebLogic Server.

2. Override the `NavTreeExtensionBacking.getTreeExtension( PageBackingContext ppCtx, String extensionUrl)` method.

   In your implementation of this method:

   a. Construct a `com.bea.jsptools.tree.TreeNode` object for the parent node.

      Use the following constructor:
      `TreeNode(String` *`nodeId`*`, String` *`nodeName`*`, String` *`nodeUrl`*`)`
      where:

      `nodeId` is the value of the control's `definitionLabel`. You can use `PageBackingContext.getDefinitionLabel()` to get this value. Alternatively, you can enter the `definitionLabel` value that is in the control's `.pinc` file.

      `nodeName` is the text that you want to display in the NavTreePortlet. You can create a `String` object that contains the text or use `PageBackingContext.getTitle()` to get this value from the page's `.pinc` file.

Note: The `PageBackingContext.getTitle()` method returns the literal value of the `title` attribute in the `.pinc` file; it never assumes that this value is a key and therefore never attempts to look up the value from a message bundle. If your `NavTreeExtensionBacking` class needs to support localization, include logic in your class to look up the locale, use the `PageBackingContext.getTitle()` method to get the `title` value, and then look up the corresponding value from the message bundle. For an example of such logic, see Listing 6-5, which localizes a portlet title (instead of a page title).

`nodeURL` is a URL to the control. Supply *`extensionUrl`* as the value of this parameter.

b. If you want to add a tree of nodes, construct additional `TreeNode` objects as children of the parent `TreeNode`.

   For each child node, use the following constructor:

   ```
   TreeNode(String nodeId, String nodeName,
      String nodeUrl, TreeNode parent)
   ```

   where:

   `nodeId` is the value of the control's `definitionLabel`. You can **not** use `PageBackingContext.getDefinitionLabel()` to get this value because the `PageBackingContext` available to this method is for the parent node. Instead, you must enter the `definitionLabel` value that is in the control's `.pinc` file.

   `nodeName` is the text that you want to display in the NavTreePortlet.

   `nodeURL` is a URL to the control. Supply the following value:

   `/console/console.portal?_nfpb=true&_pageLabel=`***`definitionLabel`***

   where *`definitionLabel`* is the `definitionLabel` of the page to which you want to link.

   `parent` is any `TreeNode` that you have constructed. You can create multiple levels in your node tree by specifying a parent that is a child of node higher up in the hierarchy.

c. Pass the parent `TreeNode` object to the constructor for `com.bea.console.utils.NavTreeExtensionEvent`.

   Use the following constructor:

   ```
   NavTreeExtensionEvent(String pageLabel, String url,
      String parentPath, TreeNode node, int ACTION)
   ```

   where:

pageLabel is the same *nodeID* value that you used when constructing the `TreeNode` object for the parent node.

url is the same *nodeURL* value that you used when constructing the `TreeNode` object for the parent node.

parentPath is the name of the node under which you want your node to display. Use `/` (slash) to represent the root of the navigation tree in the NavTreePortlet.

For example, if you want your node or node tree to display at the top level, specify `/`. If you want your node to display as a child of Environments, specify `/Environments`.

node is the parent TreeNode that you created in step a.

`ACTION` is `NavTreeExtensionEvent.APPEND_ACTION`. For information about other possible actions, see `NavTreeExtensionEvent` in the *WebLogic Server Administration Console API Reference*.

d. Return the `NavTreeExtensionEvent` object that you constructed.

3. Save the compiled class in a package structure under your extension's `WEB-INF/classes` directory.

Listing 6-9 Example NavTreeExtensionBacking Class

```
package com.mycompany.consoleext;

import com.bea.netuix.servlets.controls.page.PageBackingContext;
import com.bea.jsptools.tree.TreeNode;
import com.bea.console.utils.NavTreeExtensionBacking;
import com.bea.console.utils.NavTreeExtensionEvent;

public class CustomNavTreeExtension extends NavTreeExtensionBacking {

   public NavTreeExtensionEvent getTreeExtension(PageBackingContext ppCtx,
      String extensionUrl){
     /*
      * Construct a TreeNode for the control that has invoked this method.
      */
      TreeNode node = new TreeNode(ppCtx.getDefinitionLabel(),
         ppCtx.getTitle(),extensionUrl);

     /*
      * Construct a child TreeNode.
      */
      TreeNode node1 = new TreeNode("MyAppGeneralTabPage",
         "MyApp General",
         "/console/console.portal?_nfpb=true&_pageLabel=MyAppGeneralTabPage",
           node);

     /*
      * Add the parent node (which includes its child) below the
      * Environment node in the NavTreePortlet.
      */
      NavTreeExtensionEvent evt =
        new NavTreeExtensionEvent(ppCtx.getDefinitionLabel(),extensionUrl,
        "/Environment",node);

      return evt;
   }
}
```

Invoke the NavTreeBacking Class

To invoke the NavTreeBacking class and start the process described in "Example: How a NavTreeExtensionBacking Class Adds a Node Tree to the NavTreePortlet" on page 6-28:

1. Determine which UI page control you want to add as the parent node.

   Only page controls can be added as nodes to the NavTreePortlet.

2. Add the following attribute and attribute value to the control's `netuix:page` element in the control's `.pinc` file:
   `backingFile="`*`your-NavTreeBacking-class`*`"`

   where *`your-NavTreeBacking-class`* is the fully-qualified name of the class you created in step 1.

Example: How a NavTreeExtensionBacking Class Adds a Node Tree to the NavTreePortlet

The following example describes how a `NavTreeExtensionBacking` class adds the node tree illustrated in Figure 6-4:

1. As the portal framework loads your extension, it parses your extension's `.pinc` files and finds a `netuix:page` element.

   For example:

   ```
   <netuix:page definitionLabel="MyAppTablePage" title="My App"
      markupName="page"
      markupType="Page"
      backingFile="com.mycompany.utils.MyNavTreeExtension"
   >
   ```

2. The portal framework instantiates a `com.bea.netuix.servlets.controls.page.PageBackingContext` object, which is an in-memory representation of the page UI control. The object contains properties that describe the page control's `title` and `definitionLabel` among other data.

3. When the portal framework encounters the `backingFile` attribute in the `netuix:page` element, it initializes the specified class (`MyNavTreeExtension`) and passes your page's `PageBackingContext` object to the class constructor. It also passes a `String` object that contains the page control's URI.

4. The `MyNavTreeExtension` class does the following:

   a. It retrieves the `title` and `definitionLabel` values from the `PageBackingContext` object.

   b. It constructs a `com.bea.jsptools.tree.TreeNode` object and passes the `title` and `definitionLabel` values along with the page control's URI to the constructor.

   c. It constructs two additional `TreeNode` objects for two pages whose titles are "Monitor EJBs" and "Log Messages."

      Because there is no way to retrieve the `PageBackingContext` objects or the URIs for these two pages, the values must be hard-coded in the `MyNavTreeExtension` class.

      To make the pages into child nodes of the "My App" page node, the `MyNavTreeExtension` class uses a form of the `TreeNode` constructor that accepts the name of a parent node. For example:

      ```
      TreeNode childnode1 = new TreeNode("MyAppMonitorEJB",
        "Monitor EJBs",
        "/console/console.portal?_nfpb=true&_pageLabel=MyAppMonitorEJB",
        node);
      ```

   d. It constructs and returns a `com.bea.console.utils.NavTreeExtensionEvent` object.

      The `NavTreeExtensionEvent` object describes the `TreeNode` objects that you constructed and indicates the location in the existing navigation tree at which you want to append your node tree.

5. The NavTreePortlet listens for `NavTreeExtensionEvent` objects. As the portlet initializes its tree, it appends nodes as specified by any `NavTreeExtensionEvent` objects that are broadcast.

Figure 6-4 Example: Adding a Node Tree to the NavTreePortlet

Navigating to a Custom Security Provider Page

If you created a custom security provider and used WebLogic MBeanMaker to create MBeans to manage your provider, the Administration Console automatically generates pages to display the provider's configuration data. It also generates a link to your provider pages from the Security: Providers table.

However, you can create your own pages to customize this display. If you create your own pages, you need to redirect the link in the Security: Providers table from the pages that the Administration Console generates to your custom pages.

To redirect the link, include the following element as a child of your page's `<netuix:page>` element:

```
<netuix:meta type="configuration" content="MBean-class-name"/>
```

where *`MBean-class-name`* is the fully qualified name of your provider's MBean class.

For example:

```
<netuix:page markupName="page" markupType="Page"
   definitionLabel="SimpleSampleAuthorizerAuthorizerConfigCommonTabPage"
   title="tab.common.label"
   skeletonUri="/framework/skeletons/default/wlsworkspace
      /page_content.jsp">
   <netuix:meta name="configuration"
   content="examples.security.providers.authorization.simple.
      SimpleSampleAuthorizerMBean"/>
   <netuix:content>
...
```

CHAPTER 7

Using BEA Templates and JSP Tags

This section describes how to add a portlet that uses the Administration Console's JSP templates, styles, and user input controls. For example, you can add portlets that render your content as one of the following:

- A table in the `ContentBook` that summarizes the resources you have provided and that enables users to navigate to a specific resource or to invoke actions on the resource from the table. (See Figure 2-5 for an example of a WebLogic Server table.)
- A form in the `ContentBook` that enables users to monitor or configure resources that you have provided.

Figure 7-1 illustrates the process. The steps in the process, and the results of each are described in Table 7-1. Subsequent sections detail each step in the process.

Figure 7-1 Administration Console Extension Development Overview

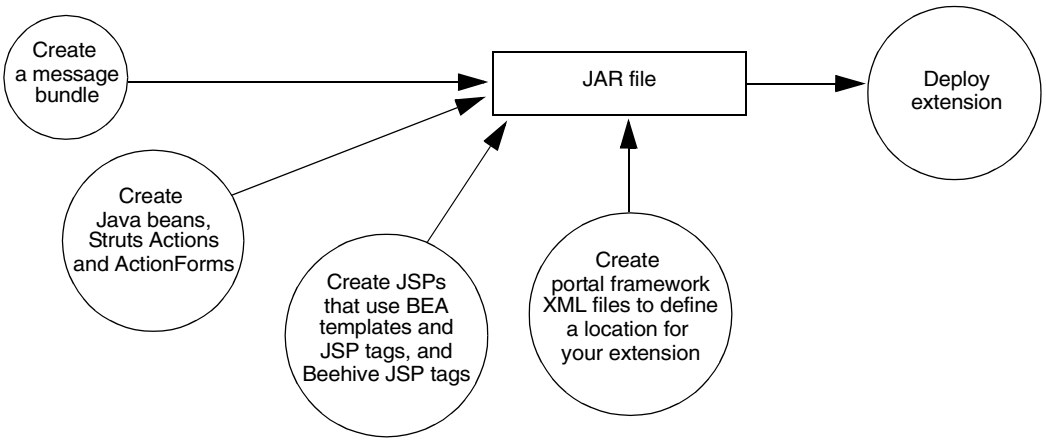


Table 7-1 Model MBean Development Tasks and Results

Step	Description	Result
1. "Create and Use a Message Bundle in Your JSPs" on page 7-3.	Create a text file that contains a name/value pair for each text string that you want to display in your extension.	One or more `.properties` files.
2. "Create Struts Artifacts for Tables and Forms" on page 7-7.	The WebLogic Server JSP tags that render forms and tables assume that Apache Struts is the controller agent. The JSP tags use Java beans that are populated by Struts `ActionForms` (form beans) and submit user input to a Struts `Action`.	A Struts configuration file, Java beans, and Java classes that implement `org.apache.struts.action.ActionForm` and `org.apache.struts.action.Action`.
3. "Create JSPs that Use BEA Templates and JSP Tags" on page 7-17.	WebLogic Server provides JSP templates that you can import into your JSPs. It also provides a JSP tag library to render the same UI controls that the Administration Console uses.	JSPs that match the Administration Console styles and structure.

Table 7-1 Model MBean Development Tasks and Results

Step	Description	Result
5."Create Other Portal Framework Files and Deploy the Extension" on page 7-37.	Create XML files that define a location for your extension.	A `.portlet` XML file that defines a portlet and configures it to launch a Struts `Action`. A `.pinc` XML file that defines a page or book control (optional), a `netuix-extension.xml` file that describes where to locate your extension, and a JAR file that automatically deploys.
6. Archive and deploy the extension.	See "Archiving and Deploying Console Extensions" on page 8-1.	A JAR file that contains your extension.

Create and Use a Message Bundle in Your JSPs

BEA recommends that you define all of the text strings that your JSPs display in a message bundle. For information about creating a message bundle, see "Creating a Message Bundle" on page 4-1.

To use the bundle in your JSPs:

1. Import the JSTL `fmt.tld` tag library:

   ```
   <%@ taglib uri="/WEB-INF/fmt.tld" prefix="fmt" %>
   ```

2. Declare the name of your bundle:

   ```
   <fmt:setBundle basename="bundle" var="current_bundle" scope="page"/>
   ```

   where *`bundle`* is the name of your bundle.

3. When you want the JSP to output a string, use the following JSP tag:

   ```
   <fmt:message key="property-name" bundle="${current_bundle}"/>
   ```

   For example:

   ```
   <fmt:message key="myextension.myTab.introduction"
   bundle="${current_bundle}"/>
   ```

Overview of Forms and Tables

WebLogic Server provides a `<wl:form>` JSP tag that can render a variety of HTML input controls, such as text controls, check boxes, and radio controls. You can configure a form to be read-only or to allow user input. Forms that allow user input must include buttons that enable users to post the form data for processing in the business layer.

WebLogic Server provides a `<wl:table>` JSP tag that renders data in a tabular format. Each row in the table represents a single entity such as a WebLogic Server instance, an application, or a log message (see Figure 7-2). You can configure table columns to render hypertext links, which enable users to navigate to pages that provide more information about an item in the table. You can also create a table column that contains an HTML check box control. If a user selects a check box for a table row and clicks a submit button, your extension can invoke business logic on behalf of the entire row. For example, you can use a check box to delete an item that a row represents.

Both of these tags use Apache Struts `Actions` and `ActionForms` to pass data between the business layer and the presentation layer.

Data Models for Forms and Tables

Apache Struts supports multiple techniques for instantiating and populating `ActionForm` beans (form beans). For example, you can code your own concrete Java bean that contains getter and setter methods for each property in the form. Or you can use the Struts `DynaActionForm` bean, which dynamically configures a Java bean to contain the properties that are declared in the Struts configuration file.

Data Model for Forms

If you are using BEA JSP tags to render a form in the Administration Console, you can use any technique for creating and populating form beans that Struts supports. (The example in "Example: How Struts Portlets Display Content" on page 2-15 uses a `DynaActionForm` bean instead of coding a custom Java bean.)

Regardless of the technique that you choose, your Java bean must contain the following property:

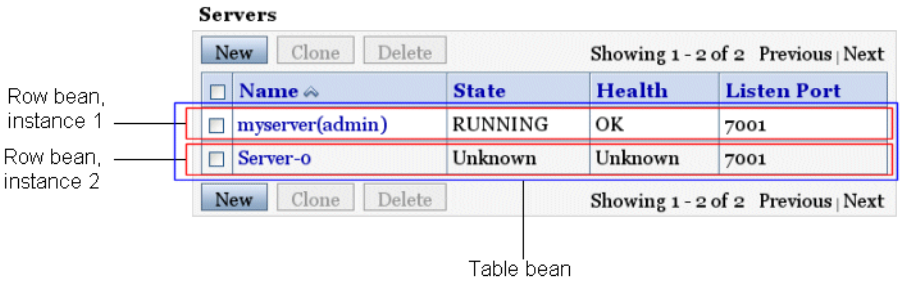
- `handle`, which can be of type `com.bea.console.handles.Handle` or a custom `Handle` class that you create.

  The portal framework uses this property to correlate an `ActionForm` bean with the data source that populates the bean, such as an MBean. See "Handles for ActionForms and Row Beans" on page 7-6.

Data Model for Tables

If you are using BEA JSP tags to render a table in the Administration Console, you must create two form beans: one bean that represents the rows in the table (called a row bean) and another bean (called a table bean) that contains the collection of row beans. Each property in the row bean is rendered as a table column. For example, in Figure 7-2, each row bean instance contains a `name`, `state`, `health`, and `listenPort` property.

Figure 7-2 Row Beans and Table Bean



To create a row bean, you must create a concrete Java bean that defines each property. You cannot use the Struts `DynaActionForm` bean to dynamically contain the properties that are declared in the Struts configuration file.

To create a table bean, you can use any technique for creating and populating form beans that Struts supports. Regardless of the technique that you choose, your table bean must contain the following properties:

- `content`, which must be of type `java.util.Collection`

  This is the property that you must use to contain the row beans.

- `handle`, which can be of type `com.bea.console.handles.Handle` or a custom `Handle` class that you create.

  While the portal framework requires you to declare this property for form beans and table beans, its usefulness is limited with table beans. Typically, a table bean is simply a collection of row beans; the row beans expose an underlying data source but the table bean does not. Unless you need to keep track of which `Action` class has populated your table bean, you do not need to set the value of this property of the table bean (but you must declare it). See "Handles for ActionForms and Row Beans" on page 7-6.

If you configure your table to include a column of check boxes, which enables you to invoke a Struts `Action` on the selected table row beans, your table bean must also contain the following property:

- `chosenContents`, which can be an array of any primitive type or an array of `com.bea.console.handles.Handle`. For information on how to work with check boxes in a table, see "Add Buttons and Checkboxes to Tables" on page 7-30.

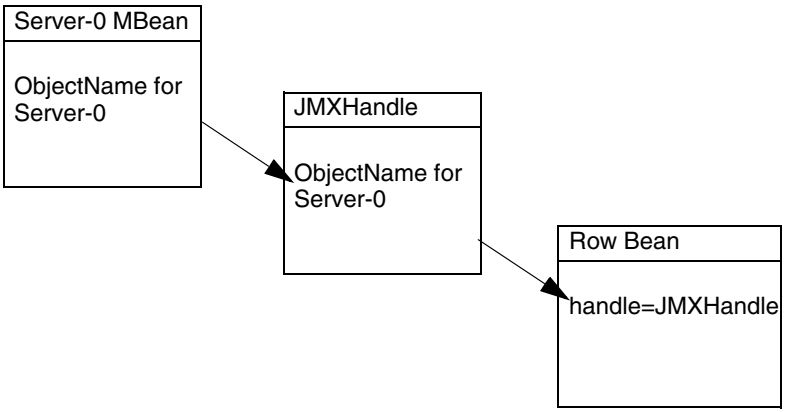
Handles for ActionForms and Row Beans

To uniquely identify an instance of an `ActionForm` bean or a row bean and to establish a correlation between the bean and its underlying data source, you can create and use a `Handle` object. A `Handle` object is a Java object that implements the `com.bea.console.handles.Handle` interface.

The Apache Struts controller servlet places `Handle` objects in `HttpServletRequest` objects, thus making them available to any Struts Action, Beehive Page Flow, or JSP.

The Administration Console uses `Handle` objects when linking from a row in a table JSP (see Figure 7-2) to the corresponding configuration JSP. For example, for a `ServerMBean` instance named Server-0, the Administration Console populates row bean with data from the Server-0 MBean. The Administration Console passes the JMX object name for Server-0 to a new a `Handle` object (of type `com.bea.console.handles.JMXHandle`) and sets the `Handle` object as the value of the row bean's `handle` property (see Figure 7-3). When a user clicks a link in the table JSP, the Struts controller looks in the row bean's `handle` property, uses the `handle` value to determine which server instance has been selected, and displays the configuration page for the selected server.

Figure 7-3 JMXHandle in a Row Bean



If the underlying data source for your `ActionForm` beans or row beans is an MBean, you can use the `com.bea.console.handles.JMXHandle` object. See `JMXHandle` in the *Administration Console API Reference*.

If the underlying data source for your beans is not an MBean, you can create your own Java class that implements the `com.bea.console.handles.Handle` interface. See `Handle` in the *Administration Console API Reference*.

Create Struts Artifacts for Tables and Forms

To render HTML forms and tables and populate them with data, the Administration Console uses JSP tags that load data from Java beans. Most of these beans contain data that a Struts `Action` has loaded from a WebLogic Server MBean. To submit user input, the JSP tags forward to Struts `Actions`, and most of these `Actions` update data in a WebLogic Server MBean.

If you use Administration Console JSP tags, you must create your own Struts `ActionForms` and `Actions`.

The following sections describe creating Java beans, Struts `Actions`, and `ActionForms` to use with forms and tables:

- "Create Struts Artifacts for a Form JSP: Main Steps" on page 7-8
- "Create Struts Artifacts for a Table JSP" on page 7-14

For information on Apache Struts, see *The Apache Struts Web Application Framework* at http://struts.apache.org/.

Create Struts Artifacts for a Form JSP: Main Steps

To create Struts artifacts that pass data between the business layer and a JSP in the presentation layer:

1. Create an `org.apache.struts.action.Action` class that populates a Java bean (form bean) with data from your business layer.

   If your form allows user input, create another `Action` class to process the data that users post from the form.

   See "Create Struts Action Classes for Handling Form Data" on page 7-8.

2. In your Struts configuration file:

   a. Declare the name and the properties of the form bean that your `Action` classes will populate and use.

      If your form allows user input, you can use the same form bean to populate the form and to return user input to your `Action` class that processes data.

   b. Create an `Action` mapping that the Struts controller uses to instantiate your form bean and invoke your `Action` class that populates the form.

      If your form allows user input, create another `Action` mapping that the Struts controller uses when users submit the form.

   See "Configure Struts ActionForms and Action Mappings" on page 7-12.

Create Struts Action Classes for Handling Form Data

To create Struts `Action` classes that handle form data:

1. Create an `org.apache.struts.action.Action` class that populates the form bean. (See Listing 7-1.)

   The Struts controller passes an empty `ActionForm` bean to your `Action` class. To populate the bean, implement the following method:

```
Action.execute(ActionMapping actionMapping,
   ActionForm actionForm,
   HttpServletRequest httpServletRequest,
   HttpServletResponse httpServletResponse)
```

Your implementation should:

a. Gather data from an underlying source, such as an MBean.

b. Cast the empty `ActionForm` bean as a `DynaActionForm` bean.

c. Invoke the `DynaActionForm.set()` method for each property that you defined in the `<form-bean>` element, except for the `handle` property.

   For example, if you defined two properties named `name` and `totalRx`:

```
DynaActionForm form = (DynaActionForm) actionForm;
form.set("name",namefromMBean);
form.set("totalRx",totalRxfromMBean);
```

d. To establish a correlation between the form bean and its underlying data source, set the value of the `handle` property. (See "Handles for ActionForms and Row Beans" on page 7-6.)

   For example, if your underlying data source is an MBean, use `JMXHandle` and set the the `handle` property to the MBean's `ObjectName`:

```
ObjectName anMBean = new
    ObjectName("com.bea.medrec:Type=com.bea.medrec.controller.
              RecordSessionEJBMBean,Name=MedRecEAR");
form.setHandle(new JMXHandle(anMBean));
```

e. Put the `DynaActionForm` bean into the request object that was also passed to the class:

```
httpServletRequest.setAttribute("form-bean-name", form);
```

   where ***`form-bean-name`*** matches the name that you configure for the form bean in the Struts configuration file (see "Configure Struts ActionForms and Action Mappings" on page 7-12).

f. Return "success" in the `ActionMapping.findForward()` method for the `ActionMapping` object that was passed to the `Action` class:

```
return actionMapping.findForward("success");
```

2. If your form posts data for processing in the business layer, create another `Action` class that processes the form data.

   When a user posts data from the form (by clicking an HTML button), the Struts controller passes a populated `ActionForm` bean to your `Action` class. To process the data, implement the following method:

   ```
   Action.execute(ActionMapping actionMapping,
      ActionForm actionForm,
      HttpServletRequest httpServletRequest,
      HttpServletResponse httpServletResponse)
   ```

   Your implementation should:

   a. Cast the `ActionForm` bean that was passed in the request as a `DynaActionForm` bean.

   b. Invoke the `DynaActionForm.get()` method for each property that you want to process.

      For example, if you want to process the properties named `name`, `totalRx`, and `handle`:

      ```
      DynaActionForm form = (DynaActionForm) actionForm;
      String nameValue = (String) form.get(namefromMBean);
      Integer totalValue = (Integer) form.get(totalRxfromMBean);
      JMXHandle handle = (JMXHandle) form.get(handle);
      ```

   c. Process the data.

      For example, if the `name` and `totalRx` properties represent attributes in a MBean and you want to change the values of the MBean attributes, use the `handle` property to get the JMX object name of the MBean instance, and then use JMX APIs to set the MBean attributes to the values that were posted from the form:

      ```
      ObjectName oName = handle.getObjectName();
      MBeanServer.setAttribute(oName, new Attribute(("Name", nameValue));
      MBeanServer.setAttribute(oName, new Attribute(("TotalRx",
      totalValue));
      ```

   d. Return "success" in the `ActionMapping.findForward()` method for the `ActionMapping` object that was passed to the `Action` class:

      ```
      return actionMapping.findForward("success");
      ```

3. Compile the Action classes and save them in a package structure that begins in the *`root-dir`*`/WEB-INF/classes` directory.

Listing 7-1 is an example `org.apache.struts.action.Action` class that accesses a custom MBean and uses it to populate a form bean.

Listing 7-1 Example: Action Class that Populates a Form Bean

```
import javax.management.MBeanServer;
import javax.management.ObjectName;
import javax.management.MalformedObjectNameException;
import javax.naming.InitialContext;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.DynaActionForm;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class MedRecMBeanFormAction extends Action {
   public ActionForward execute(ActionMapping actionMapping,
                                ActionForm actionForm,
                                HttpServletRequest httpServletRequest,
                                HttpServletResponse httpServletResponse)
      throws Exception {
      try {
         // Establish a local connection to the Runtime MBean Server
         InitialContext ctx = new InitialContext();
         MBeanServer server =
            (MBeanServer) ctx.lookup("java:comp/env/jmx/runtime");
         // Create an ObjectName that corresponds to a custom MBean that
         // has been registered in the Runtime MBean Server
         ObjectName anMBean = new ObjectName(
            "com.bea.medrec:Type=com.bea.medrec.controller.
              RecordSessionEJBMBean,Name=MedRecEAR");
         //Get the value of the custom MBean's "Name" attribute
         String namefromMBean = (String)server.getAttribute
            (anMBean, "Name");
        // Get the value of the custom MBean's "TotalRx" attribute
         Integer totalRxfromMBean = (Integer) server.getAttribute
            (anMBean, "TotalRx");

        // Populate the form bean
        DynaActionForm form = (DynaActionForm) actionForm;
        form.set("name",namefromMBean);
        form.set("totalRx",totalRxfromMBean);
        form.set("handle",(new JMXHandle(anMBean)));

        // Set the form bean in request. The name of the
        // form bean must match the "form-bean" name in your
        // Struts configuration file
        httpServletRequest.setAttribute("medrecMBeanEJBForm", form);
      } catch (Exception ex) {
```

```
            ex.printStackTrace();
        }
        return actionMapping.findForward("success");
    }
}
```

Configure Struts ActionForms and Action Mappings

To create a Struts configuration file that declares your `ActionForms` and `Action` mappings:

1. Copy the code from Listing 7-2 and paste it into the configuration file for your Struts module.

   If you have not already created a configuration file, create a text file in *`root-dir`*`/WEB-INF` (see "Create a Directory Tree for the Extension" on page 3-2). Name the file `struts-auto-config-`*`module`*`.xml`

   where *`module`* is a name that you have chosen for your Struts module. Consider using the name of your company to avoid possible naming conflicts. You must create your own Struts module; the default Struts module is reserved for BEA Actions and ActionForms. For information about Struts modules, see the Apache Struts *User Guide* at http://struts.apache.org/struts-doc-1.2.x/userGuide/index.html.

2. To configure a form bean that Struts will use to transfer data from the business layer to the JSP in the presentation layer, replace the following value in Listing 7-2:

   – ***`form-bean-name`***, a unique name that you assign to this instance of a `DynaActionForm` bean. Your `Action` class will refer to this bean name when it populates the bean and returns it to the Struts controller.

     Use a name that reflects the name of the Struts `Action` that you will use to populate the bean instance.

3. To configure an `Action` mapping that Struts will use to populate the form bean, serialize the bean, put it into an HTTP request, and forward the request to a JSP, replace the following values in Listing 7-2:

   – ***`action-name`***, a unique name that you assign to this `Action` mapping. Your `.portlet` file will refer to this `Action` name.

   – ***`custom-Action-class`***, the fully qualified name of a Java class that you create to populate the form bean. Step 5 describes how to create this class.

   – ***`form-jsp.jsp`***, the name of a JSP that you create to render the form. See "Create a Form JSP" on page 7-21.

4. If your form posts data for processing in the business layer, create another `<action>` element that specifies a custom class that you will create to process the form data.

   For ***form-bean-name*** of this second `<action>` element, you can use the same form bean that initially populated the form. If you want to post only a subset of the data for processing, instead of using the same form bean you can configure another one that defines only the properties that you want to process.

   Upon success, this additional `<action>` element can forward to the `Action` mapping that you configured in the previous step. This reloads the JSP with the updated data.

Listing 7-2 Template for Struts Configuration File

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
   "struts-config_1_1.dtd">

<struts-config>
   <form-beans>
      <form-bean name="form-bean-name"
        type="org.apache.struts.action.DynaActionForm">
        <form-property name="handle" type="com.bea.console.handles.Handle"/>
         <!-- insert additional "form-property" elements here -->
      </form-bean>
   </form-beans>

   <action-mappings>
      <action path="/action-name"
         type="custom-Action-class"
         name="form-bean-name"
         scope="request"
         validate="false">
         <forward name="success" contextRelative="true"
            path="/ext_jsp/form-jsp.jsp"/>
      </action>
      <!-- insert additional "action" elements here -->
   </action-mappings>
   <message-resources parameter="global"/>
   <message-resources parameter="validationmessages" key="VALIDATIONMESSAGES"/>
   <message-resources parameter="genresources" key="GENRESOURCES"/>
   <message-resources parameter="global" key="GLOBAL"/>
</struts-config>
```

Create Struts Artifacts for a Table JSP

To create a Java row bean, Struts `Action`, and `ActionForm` for a JSP that uses the WebLogic Server `<wl:table>` JSP tag:

1. To configure a bean that will function as the row bean, create a standard Java bean that contains one property for each data item that you want to display in the table.

   Compile your Java bean and save it in a package structure that begins in the *`root-dir`*`/WEB-INF/classes` directory.

2. To configure a bean that will function as the table bean:

   a. Copy the code from Listing 7-2 and paste it into the configuration file for your Struts module.

      If you have not already created a configuration file, create a text file in *`root-dir`*`/WEB-INF` (see "Create a Directory Tree for the Extension" on page 3-2). Name the file `struts-auto-config-`*`module`*`.xml`

      where *`module`* is a name that you have chosen for your Struts module. Consider using the name of your company to avoid possible naming conflicts. You must create your own Struts module; the default Struts module is reserved for BEA Actions and ActionForms. For information about Struts modules, see the Apache Struts *User Guide* at http://struts.apache.org/userGuide/index.html.

   b. Replace the following value in Listing 7-2:

      `form-bean-name`, a unique name that you assign to this instance of a `DynaActionForm` bean. Your `Action` class will refer to this bean name when it populates the bean and returns it to the Struts controller.

      Use a name that reflects the name of the Struts `Action` that you will use to populate the bean instance.

   c. Add the following property:

      ```
      <form-property name="contents" type="java.util.Collection"/>
      ```

      This property will contain the collection of row beans that your `Action` class instantiates and populates.

3. To configure an `Action` mapping that Struts will use to populate the row beans and the table bean, serialize the beans, put them into an HTTP request, and forward the request to a JSP, replace the following values in Listing 7-2:
   - ***`action-name`***, a unique name that you assign to this `Action` mapping. Your `.portlet` file will refer to this `Action` name.
   - ***`custom-Action-class`***, the fully qualified name of a Java class that you create to populate the row beans and table bean. Step 5 describes how to create this class.
   - ***`form-jsp.jsp`***, the name of a JSP that you create to render the table. See "Create a Form JSP" on page 7-21.
4. Create an `org.apache.struts.action.Action` class that populates the row beans and table bean. (See Listing 7-3.)

   To populate the beans, implement the following method:

   ```
   Action.execute(ActionMapping actionMapping,
      ActionForm actionForm,
      HttpServletRequest httpServletRequest,
      HttpServletResponse httpServletResponse)
   ```

   Your implementation should:

   a. Gather application from underlying data sources, such as instances of an MBean.

   b. Create instances of your row bean and populate them by invoking their setters for each property in the bean.

   c. Assign all of your row bean instances to an `ArrayList`.

   d. Cast the empty `ActionForm` bean (table bean) as a `DynaActionForm` bean.

   e. Set the table bean's `content` property to contain the `ArrayList` of row beans:

   ```
   DynaActionForm table = (DynaActionForm) actionForm;
   table.set("contents",rowBeanArray);
   ```

   f. Put the table bean into the request object that was also passed to the class:

   ```
   httpServletRequest.setAttribute("table-bean-name", table);
   ```

   where ***`table-bean-name`*** is the name that you configured for the table bean in the Struts configuration file (see Listing 7-2).

   g. Return "success" in the `ActionMapping.findForward()` method for the `ActionMapping` object that was passed to the `Action` class:

   ```
   return actionMapping.findForward("success");
   ```

5. Compile the `Action` class and save it in a package structure that begins in the *root-dir*`/WEB-INF/classes` directory.

Listing 7-3 Example: Action Class that Populates a Row Bean and a Table Bean

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.Set;

import javax.management.MBeanServer;
import javax.management.ObjectName;
import javax.management.MalformedObjectNameException;
import javax.naming.InitialContext;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.DynaActionForm;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class RetrieveCustomMBeansAction extends Action {
   public ActionForward execute(ActionMapping actionMapping,
                                ActionForm actionForm,
                                HttpServletRequest httpServletRequest,
                                HttpServletResponse httpServletResponse)
      throws Exception {
      try {
         // Establish a local connection to the Runtime MBean Server
         InitialContext ctx = new InitialContext();
         MBeanServer server =
            (MBeanServer) ctx.lookup("java:comp/env/jmx/runtime");
         // Create a name pattern for all MedRec EJB MBeans
         ObjectName namepattern = new
              ObjectName("com.bea.medrec:Type=com.bea.medrec.controller.
               RecordSessionEJBMBean,*");
         // Get all MedRec EJB MBeans for all applications
         Set objects = server.queryNames(namepattern, null);
         // Walk through each of these MBeans and get the object name
         // and the value of its TotalRX attribute
         Iterator i = objects.iterator();
         while (i.hasNext()) {
            ObjectName anMBean = (ObjectName) i.next();
            String identifier = anMBean.toString();
            Integer totalRxfromMBean =
```

```
                (Integer) server.getAttribute(anMBean, "TotalRx");
             // Instantiate a row bean.
             MedRecMBeanTableBean row = new MedRecMBeanTableBean(anMBean);
                // Set the properties of the row bean
                row.setCanonicalName(anMBean.getCanonicalName());
                row.setTotalRxinTableBean(totalRxfromMBean);
                // Add each row bean to an ArrayList
                result.add(row);
          }
       } catch (Exception ex) {
             ex.printStackTrace();
       }

       // Instantiate the table bean
       DynaActionForm form = (DynaActionForm) actionForm;
       // Set the array of row beans as the value of the table bean's "contents"
       // property
       form.set("contents",result);

       // Set the table bean in request. The name of the
       // table bean must match the "form-bean" name in your
       // Struts configuration file
       httpServletRequest.setAttribute("genericTableForm", form);
       return actionMapping.findForward("success");
   }
}
```

Create JSPs that Use BEA Templates and JSP Tags

Most portlets in the Administration Console JSPs that are based on the `tableBaseLayout_netui` and `configBaseLayout_netui` templates.

The following sections describe how to create JSPs that use these templates:

WebLogic Server JSP Templates

Table 7-2 describes the JSP templates that you can use for your Administration Console extensions. All of the templates are located in the `/layouts` directory, which is relative to the `WEB-INF` directory of the Administration Console. WebLogic Server does not publish the templates themselves, but "Using BEA Templates and JSP Tags" on page 7-1 describes how to use them.

If these templates do not meet your needs, you can create your own templates and structure the content directly in your JSP.

Table 7-2 Administration Console JSP Templates

Template	Description
`tableBaseLayout_netui.jsp`	The Administration Console uses this template for all of its JSPs that render a single table (see Figure 2-5). To create the overall structure of the document, the template outputs an HTML table with two rows. The first row contains everything in the including document's `<beehive-template:section name="configAreaIntroduction">` tag, which is usually the document's introductory text. The second row contains everything in the including document's `<beehive-template:section name="table">` tag, which is usually a table that displays a list of WebLogic Server resources and a button bar for working with the resources.

Table 7-2 Administration Console JSP Templates

Template	Description
`configBaseLayoutNoTransact.jsp`	You can use this template to render an introductory description, an HTML form, and Save button that posts the form data for processing by your custom Java classes. This template does not check for user permissions or require users to click the Lock & Edit button in the Change Center portlet. The template outputs an HTML table with four rows. The first and last rows display a Save button. The second row contains everything in the including document's `<beehive-template:section name="configAreaIntroduction">` tag, which is usually the document's introductory text. The third row contains everything in the including document's `<beehive-template:section name="form">` tag, which is a form that provides user-input controls and descriptions.
`configBaseLayout_netui.jsp`	The Administration Console uses this template for all of its JSPs that render an introductory description, an HTML form, and Save and Cancel buttons (see Figure 2-4). The template output depends on whether the user has privileges to modify the domain's configuration. If a user has permission, the template outputs an HTML table with four rows. The first and last rows display Save and Cancel buttons along with a message indicating whether the user has a lock on the configuration and can make changes. If a user does not have permission, the table does not contain these rows. The second row contains everything in the including document's `<beehive-template:section name="configAreaIntroduction">` tag, which is usually the document's introductory text. The third row contains everything in the including document's `<beehive-template:section name="form">` tag, which is a form that provides user-input controls and descriptions.

Create a Form JSP

Before you create a form JSP, create Struts artifacts that pass data between the business layer and the JSP. See "Create Struts Artifacts for a Form JSP: Main Steps" on page 7-8.

To create a form JSP (see Listing 7-4):

1. Create a JSP and save it in your development directory. Consider creating a subdirectory to contain all of the JSPs in your extension. For example, *root-dir*/jsp
   where *root-dir* is your development directory. For more information, see "Setting Up a Development Environment" on page 3-1.

2. Import JSP tag libraries by including the following tags:
   ```
   <%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
   <%@ taglib uri="/WEB-INF/console-html.tld" prefix="wl-extension" %>
   <%@ taglib uri="/WEB-INF/fmt.tld" prefix="fmt" %>
   <%@ taglib uri="/WEB-INF/beehive-netui-tags-template.tld"
   prefix="beehive-template" %>
   ```
   For information about these tag libraries, see "JSP Tag Libraries" on page 2-13.

3. (Optional) If you plan to use `<fmt:message>` tags to display localized text, use `<fmt:setBundle/>` to specify the name of the message bundle.

   This `<fmt:setBundle/>` tag enables you to specify the bundle name once, and then refer to this value from `<fmt:message>` tags by variable.

4. Declare the JSP template for configuration pages by creating the following opening tag:

   ```
   <beehive-template:template
   templatePage="/layouts/configBaseLayoutNoTransact.jsp">
   ```
   Do not close the tag yet. All other JSP tags in a form JSP are nested in this template tag.

   Note: If your form modifies attributes of WebLogic Server MBeans, use the `configBaseLayout_netui.jsp` template instead. See "JSP Tag Libraries" on page 2-13.

5. Create a `<beehive-template:section name="configAreaIntroduction">` tag. Inside this tag, provide an introductory sentence or paragraph that describes the form. This description is rendered above the form.

6. Create the following opening tag:
   ```
   <beehive-template:section name="form">
   ```
   Do not close the tag yet.

7. Indicate that the next set of JSP tags output XHTML by creating the following tag:

   ```
   <html:xhtml/>
   ```

8. Create an opening `<wl-extension:template name="/WEB-INF/templates/form.xml">` tag.

   This template creates a form that matches Administration Console configuration pages (such as Domains: Configuration: General).

   The template also generates a button that submits the form.

9. Create an opening `<wl-extension:form>` and specify values for the following attributes:

   – `action`, (optional) if your form accepts user input, specify the path of a Struts `Action` that is invoked when a user submits this form. The Struts module that defines the `Action` path is specified in the request.

   – `bundle`, (optional) specify the name of a message bundle that contains localized names of your column headings.

   – `readOnly`, (optional) specify "true" to make this form read-only (for example, if you are displaying read-only monitoring data).

10. For each property in the form bean that you want to display in the form, create a `<wl-extension>` tag corresponding to the type of control that you want to render (see *WebLogic Server JSP Tags Reference*):

    – `<wl-extension:checkbox>`

    – `<wl-extension:chooser-tag>`

    – `<wl-extension:hidden>`

    – `<wl-extension:password>`

    – `<wl-extension:radio>`

    – `<wl-extension:select>`

    – `<wl-extension:text>`

    – `<wl-extension:text-area>`

    Alternatively, you can use `<wl-extension:reflecting-fields>`, which generates an HTML input tag for each property in a form bean. For example, for a bean property that contains a `java.lang.String`, the tag generates a text control; for a boolean, it generates a checkbox. This tag uses the default form bean, which is passed to the JSP in the request.

11. If your form accepts user input and does not modify the attributes of WebLogic Server MBeans, be sure to include the `singlechange="false"` attribute in the `<wl-extension>` tags described in the previous step.

    This attribute enables users to post form data without starting a WebLogic Server edit session.

12. To generate text on the page that describes to users the purpose of each control, include the `inlineHelpId` attribute in each `<wl-extension>` tag in the previous step.

13. Close the `<wl-extension:form>`, `<beehive-template:section>`, and `<beehive-template:template>` tags.

Listing 7-4 Example: Simple Form JSP

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/console-html.tld" prefix="wl-extension" %>
<%@ taglib uri="/WEB-INF/fmt.tld" prefix="fmt" %>
<%@ taglib uri="/WEB-INF/beehive-netui-tags-template.tld"
   prefix="beehive-template" %>

<fmt:setBundle basename="mycompany" var="current_bundle" scope="page"/>
<beehive-template:template
templatePage="/layouts/configBaseLayoutNoTransact.jsp">
   <beehive-template:section name="configAreaIntroduction">
      <fmt:message key="mycompany.myresource.introduction"
           bundle="${current_bundle}"/>
   </beehive-template:section>

   <beehive-template:section name="form">
      <html:xhtml/>
      <wl-extension:template name="/WEB-INF/templates/form.xml">
         <wl-extension:form action="/MyCompanyMyResourceUpdated" bundle="core">
            <wl-extension:text property="MyResourceName"
               labelId="mycompany.myresource.name.label"
                inlineHelpId="mycompany.myresource.name.label.inlinehelp"
                singlechange="false"/>
             <wl-extension:select
                 property="MyResourceWidgets"
                 labelId="mycompany.myresource.widgets.label"
                 inlineHelpId="mycompany.myresource.widgets.label.inlinehelp"
                 singlechange="false">
                 <wl-extension:optionsCollection
                     property="MyResourceAvailableWidgets"
                     label="label" value="value"/>
```

```
            </wl-extension:select>
         </wl-extension:form>
      </wl-extension:template>
   </beehive-template:section>
</beehive-template:template>
```

Create a Table JSP for Monitoring

Before you create a table JSP, create Struts artifacts that pass data between the business layer and the JSP. See "Create Struts Artifacts for a Table JSP" on page 7-14.

To create a table JSP for monitoring resources (see Listing 7-5):

1. Create a JSP and save it in your development directory. Consider creating a subdirectory to contain all of the JSPs in your extension. For example, *root-dir*/jsp
   where *root-dir* is your development directory. For more information, see "Setting Up a Development Environment" on page 3-1.

2. Import JSP tag libraries by including the following tags:

   ```
   <%@ taglib uri="/WEB-INF/console-html.tld" prefix="wl-extension" %>
   <%@ taglib uri="/WEB-INF/fmt.tld" prefix="fmt" %>
   <%@ taglib uri="/WEB-INF/beehive-netui-tags-template.tld"
   prefix="beehive-template" %>
   ```

   For information about these tag libraries, see "JSP Tag Libraries" on page 2-13.

3. (Optional) If you plan to use `<fmt:message>` tags to display localized text, use `<fmt:setBundle/>` to specify the name of the message bundle.

   This `<fmt:setBundle/>` tag enables you to specify the bundle name once, and then refer to this value from `<fmt:message>` tags by variable.

4. Declare the JSP template for tables by creating the following opening tag:

   ```
   <beehive-template:template
      templatePage="/layouts/tableBaseLayout_netui.jsp">
   ```

   Do not close the tag yet. All other JSP tags in a table JSP are nested in this template tag.

5. Create a `<beehive-template:section name="configAreaIntroduction">` tag. Inside this tag, provide an introductory sentence or paragraph that describes the table. This description is rendered above the table.

6. Create the following opening tag:
   `<beehive-template:section name="table">`

   Do not close the tag yet.

7. Create an opening `<wl-extensions:table>` tag and specify values for the following minimal attributes:
   - `name`, specify the name of the form bean that you configured for this table.
   - `property`, specify the name of the form-bean property that contains row beans.
   - `bundle`, (optional) specify the name of a message bundle that contains localized names of your column headings.
   - `captionEnabled`, (optional) specify "true" to generate a title above the table.

8. If you specified "true" for the `captionEnabled` attribute, create a `<wl-extension:caption>` tag. Inside this tag, provide a caption for the table.

9. For each property in the row bean that you want to display in the table, create a `<wl-extension:column>` tag and specify values for the following attributes:
   - `property`, specify the name of the row bean property
   - `label`, specify a key in your message bundle to display as the column heading

10. Close the `<wl-extension:table>`, `<beehive-template:section>`, and `<beehive-template:template>` tags.

Listing 7-5 Example: Table JSP for Monitoring

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/console-html.tld" prefix="wl" %>
<%@ taglib uri="/WEB-INF/fmt.tld" prefix="fmt" %>
<%@ taglib uri="/WEB-INF/beehive-netui-tags-template.tld"
prefix="beehive-template" %>

<fmt:setBundle basename="core" var="current_bundle" scope="page"/>

<beehive-template:template templatePage="/layouts/tableBaseLayout_netui.jsp">

   <beehive-template:section name="configAreaIntroduction">
      <fmt:message key="core.server.servertable.introduction"
            bundle="${current_bundle}"/>
   </beehive-template:section>

   <beehive-template:section name="table">
      <wl-extension:table name="extensionForm"
             property="contents"
             captionEnabled="true"
            bundle="core">

         <wl-extension:caption>
            <fmt:message key="server.table.caption"
                 bundle="${current_bundle}"/>
         </wl-extension:caption>
         <wl-extension:column property="name"
            label="server.table.label.name"/>
         <wl-extension:column property="clusterName"
            label="server.table.label.cluster"/>
         <wl-extension:column property="machineName"
            label="server.table.label.machine"/>
      </wl-extension:table>

   </beehive-template:section>

</beehive-template:template>
```

Create a Table Column for Navigating to Other Pages

Your table JSP can provide a link from each row to a configuration page or some other related page. The linking mechanism uses a `Handle` object to determine which pages are related to a specific table row (see "Handles for ActionForms and Row Beans" on page 7-6).

You can use any of the following JSP tags to link from a table:

- `<wl:column-link>`, which requires you to specify the label of the page and portlet instance to which you want to link. The handle causes the portlet to display data related to the specific row that you selected.
- `<wl:column-dispatch>`, which uses metadata to determine the page and portlet to display. Instead of specifying the page and portlet label, you add a metadata tag to the page declaration and then specify the metadata value in the `<wl:column-dispatch>` tag. Using metadata enables you to change page labels without breaking links. The handle is still used to cause the portlet in the page to display data related to the specific row that you selected.

The following sections describe how to create a table column for navigating:

- "Add a Handle to Your Row Bean and Action Class" on page 7-27
- "Use the column-link Tag" on page 7-28
- "Use the column-dispatch Tag" on page 7-29

Add a Handle to Your Row Bean and Action Class

To create and populate a `handle` property:

1. In your row bean, add a property named `handle` whose data type is `com.bea.console.handles.Handle`:

   ```
   public com.bea.console.handles.Handle getHandle() {
      return handle;
   }
   public void setHandle(Handle handle) {
      this.handle = handle;
   }
   ```

2. In the Struts `Action` class that populates the row bean, set the value of the `handle` property.

   If you populate your row bean from data in an MBean, create a `com.bea.console.handles.JMXHandle` object that contains the JMX `ObjectName` of the MBean. Then set the `JMXHandle` object as the value of the `handle` property:

   ```
   javax.management.ObjectName anMBean = new
       ObjectName("com.bea.medrec:Type=com.bea.medrec.controller.
                  RecordSessionEJBMBean,Name=MedRecEAR");
   row.setHandle(new JMXHandle(anMBean));
   ```

   If your populate your row bean from some other type of data source, you can create a `JMXHandle` object by passing a `String` to the constructor instead of an `ObjectName`. The

`String` must contain the following character sequence: `Type=`*`identifier`*, where *`identifier`* is something that is meaningful to you:

```
row.setHandle(new JMXHandle("Type=myDataSource"));
```

You can also create and set a custom `Handle` object. See `Handle` in the *Administration Console API Reference*.

3. Recompile your row bean and `Action` class.

Use the column-link Tag

To use the `<wl:column-link>` tag:

1. At the top of the table JSP, add the following statement to import the `render` tag library into your table JSP:

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

   BEA provides this tag library in its runtime environment.

2. In the `<wl:table>` tag, add the following attribute:
   `CheckBoxValue="handle"`

3. In the `<wl:column>` tag that renders the column from which you want to link, nest the `<wl:column-link>` JSP tag:

   ```
   <wl:column-link portlet="portlet-instanceLabel">
      <render:pageUrl pageLabel="page-definitionLabel"/>
   </wl:column-link>
   ```

   where:

   – ***`portlet-instanceLabel`*** is the label of the portlet instance to which you want to link.

     The label is defined in the `instanceLabel` attribute of the `<netuix:portletInstance>` element, which is in the `.pinc` file for the page that contains the portlet.

   – ***`page-definitionLabel`*** is the unique label of the page that contains the instance of the portlet to which you want to link.

     The label is defined in the `definitionLabel` attribute of the `<netuix:page>` element, which is in the page's `.pinc` file.

For example:

```
<wl:column property="Name"
   label="medrecMBean.name.label">
   <wl:column-link portlet="medrecMonitorTabPortlet">
      <render:pageUrl pageLabel="medrecMonitor"/>
   </wl:column-link>
</wl:column>
```

Note: The `<render:pageUrl/>` tag is a convenience tag for generating a portal framework URL. See `<render:pageUrl> Tag` in *WebLogic Workshop Help*.

Use the column-dispatch Tag

To use the `<wl:column-dispatch>` tag:

1. In the `.pinc` file that defines the page to which you want to link, find the page's `<netuix:page>` element and nest the following element:

   ```
   <netuix:meta name="perspective-name" content="ObjectType-value"/>
   ```

   where:

   – ***`perspective-name`*** is a name that is meaningful to you. This value must match the value that you specify in the `perspective` attribute of the `<wl:column-dispatch>` tag. For example, specify `myCompany-configuration-page`.

   – ***`ObjectType-value`*** is the value of the `ObjectType` property in the row bean's `Handle` object. See `Handle.getObjectType()` in the *Administration Console API Reference*.

     For example, assume that you populate your row bean from data in an MBean. You use the MBean's `ObjectName` to construct a `JMXHandle` object and then set the object as the value of the row bean's `handle` property. If the MBean's `ObjectName` is "`com.mycompany:Name=myApp1,Type=myAppMBean`", then the value of `JMXHandle.ObjectType` is `myAppMBean`.

   For example:

   ```
   <netuix:meta name="myCompany-configuration-page" content="myAppMBean"/>
   ```

2. In the table JSP, in the `<wl:table>` tag, add the following attribute:
   `CheckBoxValue="handle"`

3. In the `<wl:column>` tag that renders the column from which you want to link, nest the `<wl:column-dispatch>` JSP tag:

   ```
   <wl:column-dispatch perspective="perspective-name"/>
   ```

   where:

   – ***perspective-name*** matches the ***perspective-name*** value that you specified in the `.pinc` file.

   For example:

   ```
   <wl:column property="Name"
      label="medrecMBean.name.label">
      <wl:column-dispatch perspective="myCompany-configuration-page"/>
      </wl:column-link>
   </wl:column>
   ```

Add Buttons and Checkboxes to Tables

In a table that you create using the `<wl:table>` tag, you can use buttons by themselves or in conjunction with a column of checkboxes or radio buttons.

When used by themselves, buttons can forward to page UI control. For example, in the WebLogic Server Servers table (see Figure 7-2), users click on a New button to launch the Create a Server assistant.

When used in conjunction with a checkbox, buttons can process data on behalf of one or more table rows. For example, if each row in your table represents an instance of a custom MBean that provides monitoring data for your application, you can enable users to select a checkbox for one or more rows and click a button that resets the values in the corresponding MBean instances.

The following sections describe adding checkboxes and buttons to tables:

- "Add Buttons to a Table" on page 7-31
- "Add Checkboxes and Buttons to a Table" on page 7-32
- "Example: How Checkboxes and Buttons Process Data" on page 7-34

Add Buttons to a Table

To add buttons to a table:

1. In the table JSP, add the following attributes to the `<wl-extension:table>` tag:

   ```
   singlechange="false"
   controlsenabled="true"
   ```

   The `controlsenabled` attribute enables the table to display buttons. The `singlechange` attribute enables users to click the button without having to lock the domain's configuration. (See *WebLogic Server JSP Tags Reference*.)

2. Immediately after the `<wl-extension:table>` opening tag, add the following tags:

   ```
   <wl:button-bar>
      <wl:button-bar-button labelid="button-label"
         pageLabel="page-definitionLabel"/>
   </wl:button-bar>
   ```

   where:

   – ***button-label*** is the text that you want to display on the button or the name of a property that you have defined in the bundle that has been declared in the JSP's `<fmt:setBundle>` element.

   – ***page-definitionLabel*** is the unique label of the page that contains the instance of the portlet to which you want to forward.

     The label is defined in the `definitionLabel` attribute of the `<netuix:page>` element, which is in the page's `.pinc` file.

     For example, to link to the Servers table page:

   ```
   <wl:button-bar>
      <wl:button-bar-button
         labelid="Servers"
         pageLabel="ServerTableBook" />
   </wl:button-bar>
   ```

Add Checkboxes and Buttons to a Table

To process data on behalf of one or more table rows, use checkboxes and a button to post the data to an HTTP request. You must also create a Struts `Action` or Page Flow that can retrieve and process the posted data:

1. To post data to an HTTP request on behalf of one or more table rows:

   a. In your Struts configuration file, add a property named `chosenContents` to the definition of the table's `ActionForm` bean.

      The data type for this property must be either an array of primitive types or of `com.bea.console.handles.Handle`.

      The `<wls:table>` tag adds one element to this array for each checkbox that is selected when the user submits the table.

      For example:

      ```
      <form-property name="chosenContents"
      type="[Lcom.bea.console.handles.Handle;"/>
      ```

   b. In the table JSP, add the following attributes to the `<wl-extension:table>` tag:

      ```
      singlechange=”false”
      controlsenabled=”true”
      showcheckboxes="true"
      checkBoxValue="property-name"
      ```

      where ***property-name*** is the name of a property in the row bean. The data type of this property must match the data type that you have declared for the `chosenContents` property.

      The `<wl:table>` tag adds the value of this row bean to the array in the table bean's `chosenContents` property.

      If you want the table to render radio buttons, which allow users to select only a single row, add the following attribute:

      ```
      singlechoice="true"
      ```

c. Immediately after the `<wl-extension:table>` opening tag, add the following tags:

```
<wl:button-bar>
   <wl:button-bar-button labelid="button-label"
   portlet="portlet-instanceLabel"
   pageLabel="page-definitionLabel"/>
</wl:button-bar>
```

where:

button-label is the text that you want to display on the button or the name of a property that you have defined in the bundle that has been declared in the JSP's `<fmt:setBundle>` element.

portlet-instanceLabel is the label of a portlet instance that contains the Struts Action or Beehive Page Flow that you want to launch when a user clicks the button. The label is defined in the `instanceLabel` attribute of the `<netuix:portletInstance>` element, which is in the `.pinc` file for the page that contains the portlet.

Instead of immediately launching an `Action` or Page Flow, you can specify a portlet that contains a JSP. The JSP can ask users for confirmation before launching an `Action` or Page Flow.

page-definitionLabel is the unique label of the page that contains the instance of the portlet to which you want to forward.

The label is defined in the `definitionLabel` attribute of the `<netuix:page>` element, which is in the page's `.pinc` file.

2. To create a Struts `Action` that can process the posted data:

   a. Create a portlet that forwards to a Struts `Action`. Make sure that the portlet's `instanceLabel` matches the value that you specified in step 1c.

   For example:

```
<netuix:portletInstance markupType="Portlet"
    instanceLabel="medrecMonitor.Tab.Portlet"
   contentUri="/portlets/medrec_monitor_tab.portlet"/>
```

For information about creating a portlet, see "Define a Portlet" on page 6-3.

b. In your Struts configuration file, define an `ActionForm` bean that contains a property named `chosenContents`. The data type for this property must be the same data type that you specified in step 1a.

   For example:

```
<form-bean name="processButtonForm"
   type="org.apache.struts.action.DynaActionForm">
   <form-property name="chosenContents"
      type="[Lcom.bea.console.handles.Handle;"/>
</form-bean>
```

c. In your Struts configuration file, define a Struts Action mapping that sends the data in the `ActionForm` bean to a Java class for processing.

   For example:

```
<action path="/ProcessButtonAction"
   type="com.bea.medrec.extension.MedrecMBeanButtonAction"
   name="processButtonForm"
   scope="request"
   validate="false">
   <forward name="success" contextRelative="true"
      path="/ext_jsp/button_view.jsp"/>
</action>
```

Example: How Checkboxes and Buttons Process Data

The following steps describe a table that correlates a table row with an underlying MBean data source and clears the values of attributes in the MBean:

1. In a table JSP, you configure the `<wl-extension:table>` tag to render checkboxes. You specify that if a user selects the checkbox for a row, the value of the row bean's `handle` property will ultimately be posted to the request object:

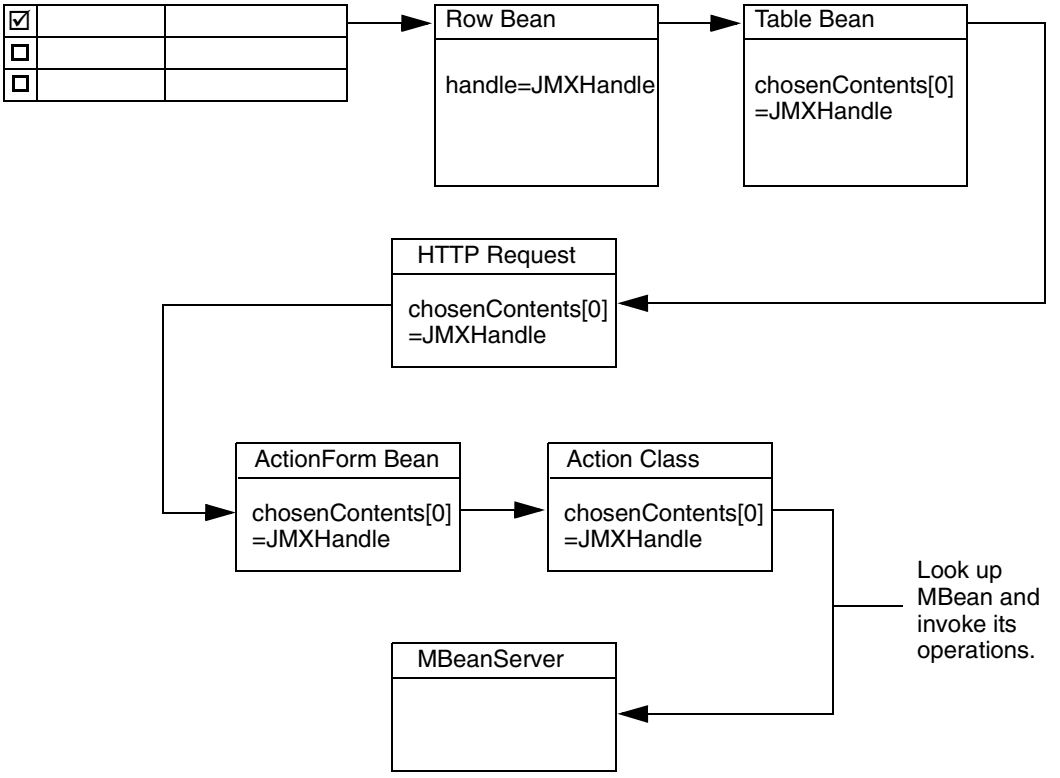
```
<wl-extension:table
    showcheckboxes="true"
    checkBoxValue="handle"
...
>
```

   The row bean's `handle` property contains a `JMXHandle` object, which contains the `ObjectName` of the MBean instance that populated the row.

2. When a user selects a row and clicks a button, the button adds the row bean's `JMXHandle` object to an array in the table bean's `chosenContents` property. Then it posts the table bean. (See Figure 7-4.)

3. The Struts controller serializes the table bean (which is a Struts `ActionForm` bean) and writes the serialized bean in the HTTP request object. Then it forwards the request to a specified portlet.
4. The portlet launches a Struts `Action` mapping, which does the following:
   a. Creates an `ActionForm` bean and populates it with data from the HTTP request.
   b. Invokes an `Action` class and makes the `ActionForm` bean available to the class.
   c. The `Action` class iterates over the form bean's `chosenContents` array (which contains instances of `JMXHandle`). For each element in the array, the class does the following:
      Gets the MBean `ObjectName` that is encoded in the `JMXHandle` object,
      Uses an `MBeanServer` to look up the MBean.
      Uses an `MBeanServer` to invoke an MBean operation that clears an attribute value.
   d. Upon success, the `Action` mapping forwards to a JSP.

Figure 7-4 Example: Data Flow from Table to Struts Action



Configure Table Preferences

By adding a single attribute to the `<wl:table>` tag, you can enable your users to configure which table columns the table displays. The Administration Console persists the preference for each user and for each instance of the portlet that displays the table. If you reuse a table in multiple portlet instances, each user can set a different preference for the table in each portlet instance.

To enable users to configure the set of table columns that your table displays, add the following attribute to your `<wl:table>` tag: `customize="true"`.

For example:

```
<wl-extension:table
    customize="true"
...
>
```

When the Administration Console displays the JSP that contains the table, it renders a "Customize this table" link above the table title. The link causes the table JSP to display a section that contains a chooser control and an Apply or Reset button.

Create Other Portal Framework Files and Deploy the Extension

You can add your portlet directly to the desktop, but if you want your portlet to display as a tab or subtab in the `ContentBook`, you must define books or pages to contain it. In addition, you must create a `netuix-extension.xml` file which specifies where to locate your portlet, books, and pages and which functions as the deployment descriptor for your extension.

See "Adding Portlets and Navigation Controls" on page 6-1.

CHAPTER 8

Archiving and Deploying Console Extensions

After you create a directory tree of source files and Java class files for your console extension, archive the directory tree in a JAR file and copy the JAR file into your WebLogic Server domain.

If you want to divide your console extensions into independently deployable components, you can create and deploy multiple archives that contain subsets of your extension. Each archive must contain all of the classes and portal framework files needed to render its own content. For example, if your extension modifies the Administration Console Look and Feel as well as adds portlets to the desktop, you can create one archive for the Look and Feel extension and another archive that contains the files needed to add your portlet to the desktop.

Archive and Deploy a Console Extension

To archive and deploy your console extension:

1. Archive your extension directory into a JAR file. The name of the JAR file has no programmatic significance, so choose a name that is descriptive for your purposes.

   The contents of your *root-dir* directory must be the root of the archive; the *root-dir* directory name itself must not be in the archive. If you use the Java `jar` command to create the archive, enter the command from the *root-dir* directory. For example:
   `c:\root-dir\> jar -cf my-extension.jar *`

2. Copy the JAR file into each domain's *domain-dir*`/console-ext` directory, where *domain-dir* is the domain's root directory.

3. Restart the Administration Server for each domain.

Error Output During Deployment

If the Administration Console encounters deployment errors, it outputs error and warning messages to standard out and to the Administration Server's server log file.

If you do not see error or warning messages and you do not see your extension in the Administration Console, you might have named the wrong parent UI control in your `netuix-extension.xml` file. For example, if you name a parent UI control that does not render tabs for its children, then your extension is deployed but there is no menu control for accessing it.