



BEA WebLogic Portal™®

White Paper: WebLogic Portal Framework

Copyright

Copyright © 2004 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

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

Trademarks or Service Marks

BEA, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Liquid Data for WebLogic, BEA Manager, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Enterprise Security, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop and How Business Becomes E-Business are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

Contents

Abstract	1
Definition of Terms, Acronyms, and Abbreviations	1
Netuix	3
Controls	3
Portal Controls	6
Desktop	6
Windows	6
Book	7
Page	7
Portlet	7
Menus	8
Layouts	8
Information for Creating a Custom Layout	8
The Layout File	9
Layout File Elements	9
Basic Layout Controls	9
Example of a Custom Layout	13
The Skeleton JSP	15
The html.txt File	17
Interacting With UI Controls	18
Context	18
Backing Context	18

Presentation Context	19
Backing Files	19
Skeletons	20
Events	20
Customization	22

Appendix

netuix-config.xml	1
Performance	4
Running Weblogic Portal Server in Production SETUP	4
Configuration in netuix-config.xml	5
Cache Configuration	5
Static Files	6
Portal Size	6
Portlet Content	6
JVM Parameter Values	6
Hardware Configurations	7
High Performing JVM parameters	7

BEA WebLogic Portal Framework

Abstract

This document is intended to give an in depth overview of the technical underpinnings of the portal framework. It is targeted towards developers who have a deep J2EE background and are already familiar with WebLogic Portal.

This document is intended to be used as a supplement to the online documentation and assumes you have already immersed your self in it and are looking for more technical substance. This document will only discuss portal framework and not directly talk about the interaction with WebLogic Workshop, the WebLogic Administration Portal or any of the other features that come with WebLogic Portal; namely: Personalization, Campaigns, Content Management, Commerce Components, Entitlements and User Management. Many features described in this document have not made their way into WebLogic Workshop or the WebLogic Administration Portal as of this writing. Nonetheless, we encourage you to explore these features.

Definition of Terms, Acronyms, and Abbreviations

It is important that we first define a set of terminology, as we will be using these terms throughout the document. Please take the time to become familiar with these concepts.

Netui

Also called Page Flows, a programming model for building model 2 type applications. Netui is built on top of the popular Struts framework.

Netuix

An XML framework for rendering applications. Netuix was originally contrived as an extension to Netui. However, today netuix is no longer based on Netui nor dependant on it. They are completely different technologies. Only the names are similar. With that said, netuix can seamlessly host netui applications.

Customization

The term used to modify a portal through an API. This API is typically called from our WebLogic Administration Portal and Visitor Tools pages but is also available to developers who wish to modify the desktop.

The API provides all the CRUD operations needed to modify a desktop and all of its components (Portlets, Books, Pages, Menus, and so on). Customization is different than Personalization. With Customization, someone is making a conscious decision to change the makeup of the desktop. With Personalization, changes are made based on rules and behavior (display an ad for Broncos tickets because it's Friday and the visitor lives in Denver).

Portal Framework

The portion of Weblogic Portal that is responsible for the rendering and *Customization* of the portal—what this document is all about.

Light Portal (File-based Portal)

A stripped down version of WebLogic Portal that does not deploy any EJBs or database. The light portal supports all the functionality in the portal framework with the exception of *Customization*. Light Portal can only render **portal** files, it cannot go to the database for a *Customized* desktop. Light portal rendering occurs in WebLogic Workshop in the development environment, but it may also be used in production systems.

UIControl

A netuix user interface control, not to be confused with business controls in WebLogic Workshop. Each element in the XML document (.portal, .portal, .shell, .layout, .laf and .menu) represents an instance of a UI control. Typical controls are Books, Pages, Menus, Portlets, and so on.

Single File vs. Streamed Rendering

The .portal file you create in WebLogic Workshop is a fully functioning portal, however, it can also be used as a template to create a desktop. In this template you create books, pages and references to portlets and define defaults for them.

When you view the .portal file with your browser the portal is rendered in “single file mode,” meaning that you are viewing the portal from your file system as opposed to a database. The .portal file's XML is parsed and the rendered portal is returned to the

browser. The creation and use of a `.portal` is intended for development purposes and for static portals (portals that are not customized by the end user or administrator). Because there is no database involved you cannot take advantage of things such as user customization or entitlements.

Note: Externally, entitlements still run, they are just difficult to set.

Once you have created a `.portal` file, you can use it to create a desktop (streamed portal). A desktop is a particular view of a portal that visitors access. Desktops can be updated by administrators and end users.

A portal can be made up of multiple desktops, making the portal a container for desktops. A desktop contains all the portlets, content, shells, layouts, and look and feel elements necessary to create individual user views of a portal. When you create a desktop based on the `.portal` file in the WebLogic Administration Portal, a desktop and its books and pages are placed into the database.

The desktop, books and pages reference shells, menus, look and feels and portlets. The settings in the `.portal` file, such as the look & feel, serve as defaults to the desktop. Once a new desktop is created from a `.portal` template, the desktop is decoupled from the template, and modifications to the `.portal` file do not affect the desktop, and vice versa.

For example, when you change a desktop's look & feel in the WebLogic Administration Portal, the change is made only to the desktop, not to the original `.portal` file. When you view a desktop with a browser it is rendered in "streaming mode" (from the database). Now that a database is involved, desktop customizations can be saved and delegated administration and entitlements can be set on portal resources.

Library

The library is a home for a set of public controls that are not associated with a desktop. In other words Books, Pages, Portlets, can be created and modified outside the scope of a desktop and then later added to a desktop. Changes to objects in the library are cascade down through the desktops and user customizations.

Netuix

Controls

As stated earlier, netuix is an XML framework for rendering applications, whether these applications look like portals or not. Many customers who use our product today create applications from our framework that look nothing like a portal. Typically when people think of

portals they think of “My Yahoo!”. While many applications developed with netuix look like My Yahoo!, many do not.

A netuix application is represented by one or more XML documents, the most familiar being the `.portal` file (an XML file with a `.portal` extension). This portal file may or may not include other portal include files, called `.pinc` files for short (files with the extension `.pinc`). Just like a JSP can include other JSP files to distribute functionality, a portal file can include other portal files.

A `.pinc` file is different from a portal file in that a portal includes the root elements or controls while the `.pinc` file does not. We will discuss this in more detail later. However, for this discussion the portal file is the parent, and it may in turn include one or more `.pinc` files, which in turn may include other `.pinc` files. One other important note: a `.pinc` file must begin with a `Book` or a `Page` element as the root element. More on what Books and Pages are in a bit.

In the portal file, you can think of each element representing an instance of a UI control. (UIControls are not to be confused with business controls in Workshop.) These controls are wired in a hierarchical tree. In other words, each control has a parent and zero or more children. The controls can discover each other at runtime and can modify the tree by adding new children or removing existing children. All controls run through a lifecycle (a set of methods called on the control in a particular order). All the methods are called in turn in a depth first order.

To best illustrate this, let’s walk through the sequence of events that happen when a person requests a portal in single file mode from the browser. But before we do that, we first need to cover a few architectural issues with the portal framework. All requests for a portal or desktop come in through the `PortalServlet`. The `PortalServlet` is registered in the `web.xml` file under the `url-patterns` **`appmanager`** and **`*.portal`**. If the `PortalServlet` detects a request ending with `.portal` it knows the request is for a locale file and does not need to go to the persistence API for the XML.

The first thing the `PortalServlet` must do is parse the XML file (`.portal`) and generate a control tree from it. Every element in the portal file represents a control in the control tree, and every attribute on the element represents an instance variable on the control. The same hierarchy is maintained in the XML document as in the control tree. A control is simply a Java class that extends another Java class, namely the `UIControl` class. In this release we don’t explicitly expose controls to developers, but developers can interact with the controls using backing files, context, and skeleton JSPs. This is discussed later.

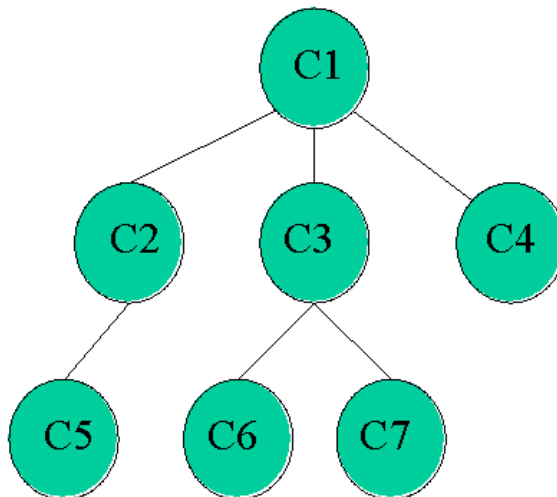
Note: The `PortalServlet` doesn’t actually parse the XML document on each request. A lot of caching and magic is going on behind the scenes to get the desired performance for the enterprise applications.

Once the control tree is built and all the instance variables are set on the controls, the control tree is run through its lifecycle. The lifecycle can be thought of as a set of methods on the controls that are called on in a well-defined order. The lifecycle methods are as follows:

```
init()  
loadState()  
handlePostBackData()  
raiseChangeEvents()  
preRender()  
saveState()  
render()  
dispose()
```

These methods are called in depth first order. In other words, all the `init()` methods are called, followed by the `loadState()` methods, and so on. They will also be called depth first. Example, given the following control tree, the order in which the `init()` method would be called is: C1, C2, C5, C3, C6, C7, C4, then the `loadState()` method would be called in the same order, and so on.

The last method to be called would be C4's `dispose()`:



Portal Controls

This section describes all the netuix controls that make up the portal framework. The control relationship is driven by the XML schema definition `controls-netuix-1_0_0.xsd`. The following figure summarizes the schema definition.

Desktop

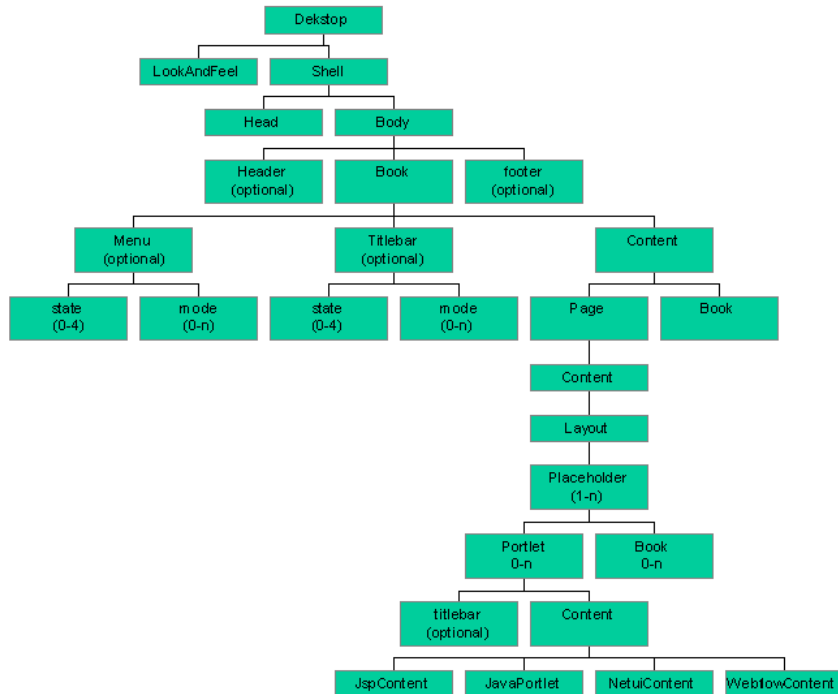
The Desktop control is the parent control that hosts all the other netuix controls. Every portal must have one Desktop control. The Desktop control actually provides little functionality above and beyond entitlement checking and a place to go to discover other controls.

The most important use of the Desktop control from a developer perspective is that it has a `PresentationContext` that can be traversed to get references to all the child controls, like books, pages, and portlets. A `DesktopBackingContext` was added in 8.1 sp3 along with a richer set of methods for locating child controls.

Windows

A Window control provides functionality similar to the windowing concept on your computer. Windows support States and Modes. States affect the rendering of the Window, like minimize, maximize, float, and delete. Modes affect the content, like Edit and Help. (Custom modes are also supported.) Windows can also act as a container for other Windows. For example, a book can contain a page.

All Window controls must have a Content control. The Content control is responsible for hosting the actual content inside the window. The Window control is an abstract class that is one of the three derived classes that must be used in the portal. These derived classes are: Books, Pages and Portlets. The figure below shows the relationship between Windows, Books, Pages and Portlets.



Book

A Book aggregates a set of navigables. A navigable is a Book or a Page. A Book may have an optional menu control that provides navigation among navigables. From a code standpoint, Navigable is an interface that Book and Page implement.

Page

A Page is used to display a set of Placeables. A Placeable is a Portlet or Book. The Page has a layout which has one or more Placeholders which can host zero or more Placeables.

Portlet

Portlets are used as windows to host many different types of applications. As of this writing the applications can be any one of the following: HTML pages, JSP files, .pinc files, Page Flows, Struts, Webflows, JSR 168 Portlets, and WSRP proxy portlets.

Menus

Menus are optional components that are loosely coupled to books and pages. A menu is responsible for displaying some type of navigation component, whether it is a set of tabs, a set of links, or some tree structure. The menu fires `PageChangeEvent`s that the Pages themselves listen to and activate accordingly.

At the time of this writing, WebLogic Portal provides two types of menus: `singlelevel` and `multilevel`. Future service packs and releases may include more. You can also create your own menus by using JSPs and the `<render:pageUrl/>` tag, or from a backing file call the `setupPageChangeEvent` method on a Book, Page or Portlet backing context before the `preRender` method.

SingleLevelMenu

Provides a single row of tabs for the book's immediate pages and child books.

MultiLevelMenu

Recursively provides a hierarchical menu for all the books and pages contained within a book. This menu does not stop at the first set of children. It continues down the tree. If the parent book uses a `multilevelmenu`, then the child books should not use a menu as the `multilevelmenu` will cover them.

Layouts

Layouts and Placeholders (not to be confused with personalization placeholders) are used to structure the way portlets and books are displayed on a page. Layout placeholders are rendered as HTML table cells.

WebLogic Portal ships with some predefined layouts and the ability to create your own custom layouts. More layouts will probably be shipped in future service packs and future releases. If the supplied layouts don't meet your needs, you will have to create your own custom layout. The next section describes that process in detail.

Information for Creating a Custom Layout

When creating a custom layout you will need to create three things:

- [The Layout File](#)
- [The html.txt File](#)

- [The Skeleton JSP](#)

The Layout File

The layout file contains the snippet of XML that describes the controls that make up the layout. The markup from this file is what gets copied into the `.portal` file and into the database for reassembly. A layout file must have a `.layout` extension and can live anywhere in the Web application directory except `WEB-INF`.

Note: Changes made to a layout file after it has been created get picked up automatically in the database but will not automatically update the layout in the `.portal` files. This is because the `.portal` file contains a copy of the markup and not a reference to it.

The `.layout` files must be created by hand (text or XML editor). The best way to get started is by copying an existing layout. Layout files shipped with the portal are located in the `/framework/markup/layout` directory.

Layout File Elements

The parent element for all markup types is:

```
<netuix:markupDefinition/>
```

This parent element has two child elements:

- `<netuix:locale language="nn" [country="nn"] [variant="nnnn"]/>`

Defines the working local for the title and description attributes defined later.

- `<netuix:markup>`

Defines the outer envelope stanza that marks the beginning and end of the XML that will define this layout.

Basic Layout Controls

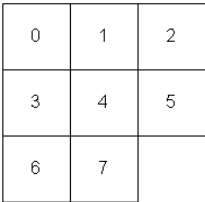
The next set of elements are unique to a layout. When creating your own layout you will have to choose from one of these four base layout controls. The `<netuix:layout/>` is the most generic of the four and all others are derived from it. The `<netuix:layout/>` control provides the most flexibility but is also the most difficult to implement.

- `<netuix:layout title="" [description=""] type = "" htmlLayoutUri="" [iconUri=""] markupName="" markupType="Layout" [skeletonUri=""] [properties=";"]/>`
- `<netuix:gridLayout columns="(1-n)" [rows = "1-n"]/>` Attributes

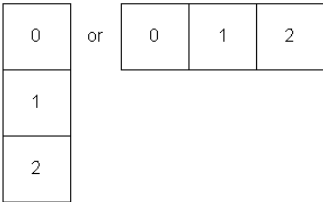
- `<netuix:borderLayout [layoutStrategy="order | title" ] />` Attributes
- `<netuix:flowLayout [orientation="vertical | horizontal"]/>` Attributes
- `<netuix:placeholder title="" [description=""] [flow="horizontal | vertical"] [usingFlow=""] [width=""] markupName="" markupType="Placeholder" [skeletonUri=""]/>`

```
<netuix:layout title="" [description=""] type = "" htmlLayoutUri="" [iconUri=""]
markupName="" markupType="Layout" [skeletonUri=""] [properties=";"]/>
```

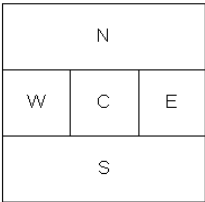
This is the base control for all layouts. This control can be used directly or you can use one of the following three derived controls.



Grid Layout
 The grid layout automatically positions the number of placeholders you specify into the number of columns and rows you specify. This examples sets columns="3" to position 8 Placeholders



Flow Layout
 The flow layout automatically positions the number of placeholders used either vertically or horizontally with no wrapping.



Border Layout
 The border layout lets you use up to five placeholders. You can position the placeholders with the attributes "north," "south," "east," "west," and "center."

Attribute	Description
title	This is the internationalized title displayed to the user and administrators when selecting the layout they want to use. Note: The developer only works in one language as defined in the <code><netuix:locale></code> element described previously. More internationalized versions of the title and description can be added later with the WebLogic Administration Portal.
description	An optional internationalized description of your layout.

Attribute	Description
type	The type of layout. This is hard coded for the three derived layouts. If you create a custom come up with your own type.
htmlLayoutUri	A fully qualified path (from the top of the Web application) to the html.txt file to be used by the WebLogic Administration Portal.
properties	“name/value pairs that can be passed to the skeleton as hints. These properties can be separated using a semicolon “;”.
iconUrl	A fully qualified path (from the top of the Web application) to the .gif file to be used by the WebLogic Administration Portal.
markType	This field is required and must be “Layout”.
markupName	This field is required and must be unique per Web application. If you copied the XML from another layout you must change this name.
skeletonUri	A fully qualified path (from the top of the Web application) to the skeleton JSP to be used for runtime rendering.
presentationClass	Optionally provides a generic presentation "class," such as a CSS class, for use by external rendering devices.
presentationStyle	Optionally provides a generic presentation "style," such as a CSS style, for use by external rendering devices.
presentationId	Optionally provides a generic presentation "id" for use by external rendering devices.

<netuix:gridLayout columns="(1-n)" [rows = "1-n"] /> Attributes

This layout defines a grid where you can specify the number of columns and rows. This layout is typically used to create one, two , three, ... column layouts.

Attribute	Description
columns	A required attribute that identifies the number of columns in the grid
rows	An optional attribute specifying the number of rows in the grid. If this attribute is omitted then the default one row will be used

`<netuix:borderLayout [layoutStrategy="order | title" ] />` Attributes

This layout has four border placeholders and one center placeholder. The north and south placeholders span the length of the table. The west, center, and east placeholders comprise the middle row and have respective widths of 25, 50, and 25 percent. The north and south portlets flow horizontally in the placeholders, and the others flow vertically.

Attribute	Description
layoutStragety	Defines what placeholder will be the north, west, center, east and south. If “title” is specified then each placeholders must specify the correct title.

`<netuix:flowLayout [orientation="vertical | horizontal"] />` Attributes

A layout that just flows the contents in a vertical or horizontal fashion.

Attribute	Description
orientation	Flow the contents vertical or horizontal. Layout controls contain one or more child placeholder controls. These controls have the following attributes.

`<netuix:placeholder title="" [description=""] [flow="horizontal | vertical"] [usingFlow=""] [width=""] markupName="" markupType="Placeholder" [skeletonUri=""] />`

Placeholders are child elements of the above four types of layouts. Every layout must have at least one placeholder child element. Each placeholder has a “layout location.” Thie loayout location is defined by its position inside the layout files. Layout locations start at 0, 1, 2.

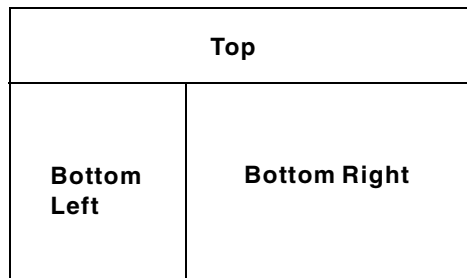
Attribute	Description
title	This is the internationalized title displayed to the user and administrators when selecting the placeholder they want to use. Note: The developer only works in one language as defined in the <code><netuix:locale></code> element described above. More internationalized versions of the title and description can be added later using the WebLogic Administration Portal.
description	An optional internationalized description of your placeholder.
markType	This field is required and must be “Placeholder”.
markupName	This field is required and must be unique per Web application. If you copied the XML from another layout you must change this name. The naming convention is <code>layoutMarkupName_placeholdersName</code> , but you can use what you want as long as it is unique.
skeletonUri	A fully qualified path (from the top of the Web application) to the skeleton JSP to be used for runtime rendering. Typically the default skeleton will suffice for all custom layouts, but you have the option to create your own.
flow	An optional value specifying the direction of content flow; default is “vertical.”
usingFlow	An optional value specifying whether or not flow should be used; default is “true.”
width	An optional hint attribute to tell the parent layout how much width this placeholder wishes to have allocated.
properties	Name/value pairs that can be passed to the skeleton as hints. These properties can be separated using a semicolon “;”. Example: <code>properties="rowspan=2;columnspan=3;myprop=hello"</code>

Example of a Custom Layout

Now that we have described the four basic layout controls and child placeholder controls, you will have to choose which one of these to base your custom layout on. Unless you are tweaking one of the parameters in the three sub-class layout controls, you may want to choose the `<netuix:layout>` control.

The easiest way to describe how to create a custom layout is to give an example. Lets create a custom layout with a spanning row at the top with two columns underneath. The two columns will split the real estate in a 30%-70% fashion.

Our layout will look something like this:



1. The first thing we need to do is create a layout file (again the easiest way is to copy one from another layout).

We will call our layout file `spanningtwocolumn.layout`, and it will look something like this:

Listing 1 Sample Code for a Layout File

```
<netuix:layout title="Spanning Two Column" description="One row and two columns."
  type="spanning"
  skeletonUri="/customskeletons/spanningtwocolumnlayout.jsp"
  htmlLayoutUri="/framework/markup/layout/spanningtwocolumn.html.txt"
  iconUri="/framework/markup/layout/spanningtwocolumn.gif"
  markupType="Layout" markupName="spanningTwoColumnLayout">
  <netuix:placeholder title="top" description="The top spanning placeholder."
    markupType="Placeholder"
    markupName="spanningTwoColumn_top">

  </netuix:placeholder>
  <netuix:placeholder title="left" description="The bottom left placeholder"
    markupType="Placeholder"
    markupName="spanningTwoColumn_left">

  </netuix:placeholder>
  <netuix:placeholder title="right" description="The bottom right placeholder"
    markupType="Placeholder"
    markupName="spanningTwoColumn_right">

  </netuix:placeholder>
</netuix:layout>
```

The `<netuix:markupDefinition>`, `<netuix:locale/>` and `<netuix:markup/>` elements were left out of the example for the sake of clarity. DON'T forget to include these in your `.layout` file.

In the previous layout example we are using the `<netuix:layout/>` element, and we have three placeholders underneath it. Other things to note are the markupNames are unique, and we have identified our own custom skeleton to do the rendering.

The Skeleton JSP

Since a custom skeleton is being used to do the rendering (as specified by the `skeletonUri` attribute) this JSP needs to be created. Again, the easiest way is to copy an existing one.

Note: The skeleton JSP for the control is called twice: once during “begin render” and once for “end render.” Between the begin render and end render phase the children are rendered. This allows you to start HTML tables in the begin render section and close them in the end render section. All skeleton files should have the following JSP tags:

- `<render:beginRender></render:beginRender>`
- `<render:endRender></render:endRender>`.

The body of the `beginRender` tag is only evaluated during the begin render phase, and the body of the `endRender` tag is only evaluated during the end render phase.

Here is what our new skeleton JSP (`/customskeletons/spanningtwocolumnlayout.jsp`) will look like

Listing 2 New Skeleton JSP

```
%@ page import="com.bea.netuix.servlets.util.RenderToolkit,
com.bea.netuix.servlets.controls.layout.LayoutPresentationContext,
java.util.List,
com.bea.netuix.servlets.controls.layout.PlaceholderPresentationContext"
%>
<%@ taglib uri="render.tld" prefix="render" %>
<%
    RenderToolkit toolkit = RenderToolkit.htmlInstance();
    LayoutPresentationContext layout =
        LayoutPresentationContext.getLayoutPresentationContext(request);
%>

<render:beginRender>
```

```

<table
  <% toolkit.writeId(out, layout.getPresentationId()); %>
  <% toolkit.writeAttribute(out, "class", layout.getPresentationClass(),
"layout-custom"); %>
  cellpadding="0"
>
  <tbody>
<%
  List children = layout.getChildren("layout:placeholder");

  // Could get optional properties here to help with rendering
  // String property = layout.getProperty("myProperty");

  for (int i = 0; i < children.size(); i++)
  {
    PlaceholderPresentationContext placeholderPresentationContext =
      (PlaceholderPresentationContext)children.get(i);
    if (i == 0)
    {
      %>
        <tr>
          <td colspan="2" width="100%" valign="top"
class="layout-placeholder-container">
            <% toolkit.renderChild(placeholderPresentationContext, request); %>
          </td>
        </tr>
      <%
        }
        else if (i == 1)
        {
      %>
        <tr>
          <td width="30%" valign="top" class="layout-placeholder-container">
            <% toolkit.renderChild(placeholderPresentationContext, request); %>
          </td>
        </tr>
      <%
        }
        else if (i == 2)
        {
      %>
        <td width="70%" valign="top" class="layout-placeholder-container">
          <% toolkit.renderChild(placeholderPresentationContext, request); %>
        </td>
        </tr>
      <%
        }
      }
    %>
  }
</render:beginRender>

<render:endRender>
  </tbody>
</table>
</render:endRender>

```

Note: This example the widths are hard coded in the JSP. Instead, these widths should be specified in the layout file as an attribute to the placeholder. The widths can then be referenced in the skeleton as follows:

```
<render:writeAttribute name="width" value="<%=
placeholderPresentationContext != null ?
placeholderPresentationContext.getWidth() : null %>"/>
```

Also, other properties like “rowspan=2” can be passed as name/value pairs on the properties attribute and “endtop” to create a more generic row/column spanning layout.

The custom layout is now functionally complete. The html.txt file has not yet been created, but the layout can be tested. To do this, start WebLogic Workshop, open or create a portal file, select a page, and in the Property Editor window select the custom layout in the Layout field.

Note: if you change your .layout file after you have used it in the .portal file, changes won't be reflected in the .portal file. This happens because when you use a layout in the .portal it copies the markup from the layout. You will need to choose another layout and then choose the original one back again to see the changes.

The html.txt File

The .html.txt is an HTML snippet strictly used by the WebLogic Administration Portal and Weblogic Workshop to give a visual representation of what the layout looks like, so the administrator can place the portlets in the correct placeholders. Typically this is the last file you will create, because it is not used by the rendering framework.

The last thing to do is create the html.txt file so the WebLogic Administration Portal can provide a visual representation of the layout. The /framework/markup/layout/spanningtwocolumn.html.txt should look something like this:

Listing 3 Sample html.txt Code

```
<table class="portalLayout" id="thePortalLayout" width="100%" height="100%">
  <tr>
    <td class="placeholderTD" valign="top" width="100%" colspan="2">
      <placeholder number="0"></placeholder>
    </td>
  </tr>
```

```
<tr>
  <td class="placeholderTD" valign="top" width="30%">
    <placeholder number="1"></placeholder>
  </td>
  <td class="placeholderTD" valign="top" width="70%">
    <placeholder number="2"></placeholder>
  </td>
</tr>
</table>
```

Interacting With UI Controls

Since controls are not exposed directly to developers, developers need a way to directly interact with and affect the behavior of the controls. To accomplish this, WebLogic Portal exposes context, backing files, skeletons, and events. Developers should use these components when trying to alter the behavior of or interact with the portal framework.

Context

A context is nothing more than a delegate to the underlying control. This delegate only exposes the supported methods on the control.

Contexts are broken down into two types: backing context and presentation context. Backing contexts are available from the backing files, and presentation contexts are available from the JSPs.

Two types of context are required because certain methods apply at certain times in the lifecycle. For example, it doesn't make sense to have a `setTitle()` method on the Presentation context because the portal has already started to render and it would have no effect. Calling this method from a backing file, however, is appropriate.

Backing Context

BackingContext is available from backing files. A reference to a Backing context can be obtained in one of two ways:

- The first way is to use the static method `getXXXBackingContext` on the context class. This method will return the *active* backing context for that type. To be more specific, if I call this method from portlet A's backing file, I will get the backing context for portlet A not portlet B.

Similarly, if I call `getPageBackingContext(request)` from portlet A, I will get the page backing context for the page portlet A is located on.

- The second way to obtain a backing context is from another context. This can be useful when you want a context that is not the *active* context. Example would be, I want to obtain portlet B's backing context from portlet A.

If portlet A is contained within the same page as Portlet B then one could use:

```
PortletBackingContext portletB =
PageBackingContext.getPageBackingContext(request).PortletBackingContext
getPortletBackingContextRecursive("Portlet Bs instance label");
```

If Portlet A does not know where portlet B is located then you can delegate to the `DesktopBackingContext`

```
PortletBackingContext portletB =
DesktopBackingContext.getPageBackingContext(request).PortletBackingCont
ext getPortletBackingContextRecursive("Portlet Bs instance label");
```

Refer to the javadoc on these and other backing context for more information.

```
com.bea.netuix.servlets.controls.page.PageBackingContext
com.bea.netuix.servlets.controls.application.backing.DesktopBackingContext
```

Presentation Context

`PresentationContext` are available from JSP files. A reference to a presentation context can be obtained in one of two ways:

- The first way is to use the static method `getXXXPresentationContext` on the context class. This method will return the *active* presentation context for that type. To be more specific, if I call this method from portlet A's content JSP, I will get the presentation context for portlet A not portlet B. Similarly, if I call `getPagePresentationContext(request)` from portlet A, I will get the page Presentation context for the page portlet A is located on.
- The second way to obtain a presentation context is from another context. This can be useful when you want a context that is not the *active* context. For example, I want to obtain portlet B's presentation context from portlet A.

Backing Files

Backing files are simple Java classes that implement the

`com.bea.netuix.servlets.controls.content.backing.JspBacking` interface or extend the `com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking` abstract class (in retrospect it should have been called a backing class). The methods on the

interface mimic the controls lifecycle methods and are invoked at the same time the controls lifecycle methods are invoked.

The controls as of this writing that support backing files are:

- Books
- Pages
- Portlets
- JspContent controls.

Note: Desktops also support backing files as of Service Pack 3

A new instance of a backing file is created per request, so you don't have to worry about thread safety issues. New Java VMs are specially tuned for short-lived objects, and this is not the performance issues it once was in the past. Also JspContent controls support a special type of backing file that allows you to specify if the backing file is thread safe. If this value is set to true, only one instance of the backing file is created and shared across all requests.

Skeletons

Skeletons are JSPs that are used during the render phase. The render phase is actually broken into two parts: begin render and end render. The parent control's begin render is called, followed by its children's begin render, their children's begin render, and so on. After the last begin render is called, the children's end renders are called, ending with the parent's end render. This allows the parent to provide a container, such as an HTML table, and the children to provide the table contents.

Each skeleton is actually called twice. There are special tags in the skeleton that only evaluate to true depending on which render phase you are in.

Events

There are four types of events in the system:

- Window Mode
- Window State
- Page Change
- Generic Portlet Events.

The Mode, State and Page Change Events are not exposed directly to the developer but can be configured through special methods on the Window backing files. Namely:

`setupModeChangeEvent`, `setupStateChangeEvent`, and `setupPageChangeEvent()`. The methods must be called before the `preRender` method as events are fired just after `handlePostBackData` method. They will also only work if the `handlePostBackData` method returns true (see [javadoc](#)).

Note: When calling one of the `setupxxevent` methods, it must be done on the backing context that the backing file is tied to. If you do not do this the event may not get fired.

Portlet Events (not to be confused with page flow events) allow portlets to communicate. One portlet can create an event and other portlets can listen for that event. These Portlet events can also carry payloads.

Here is an example of one portlet firing an event from a backing file and other portlets listening for the event:

Listing 4 Sample Code for Portlet Firing and Event from a Backing File

```
/**
 * This is the implementation on the backing file of the portlet that wants to fire the
 * event.
 */
public boolean handlePostBackData(HttpServletRequest request, HttpServletResponse
response)
{
    // Create a new portlet event with the results in the payload
    PortletEvent portletEvent = new PortletEvent(new MyPayload("Hello From portlet A"));

    // Get a hold of the portlet event manager and fire the event.
    PortletBackingContext portletBackingContext =
        PortletBackingContext.getPortletBackingContext(request);
    PortletEventManager portletEventManager =
        PortletEvent.getEventManager(this, portletBackingContext);
    portletEventManager.fireEvent(portletEvent);

    // Needed for the event to fire.
    return true;
}

/**
 * This is the implementation of the portlet that wants to receive the event.
 */
public class ResultBacking extends AbstractJspBacking implements PortletEventListener
{
    MyPayload result;
```

```

public void init(HttpServletRequest request, HttpServletResponse response)
{
    result = null;

    // Register for Portlet Events
    PortletBackingContext portletBackingContext =
        PortletBackingContext.getPortletBackingContext(request);
    PortletEvent.addGlobalListener(portletBackingContext, this);
    CustomPortletEvent.Manager portletEventManager =
        CustomPortletEvent.getEventManager(this, portletBackingContext);
}

public void handleEvent(Object source, AbstractEvent event)
{
    // Can check the source of the event
    if (source instanceof PortletA)
    {
        result = (MyPayload)((PortletEvent)event).getPayload();
    }
}

```

Note: The tutorial portal contains examples of portlets communicating via events.

Customization

Customization is the term used to describe Administrators and End Users making modifications to a desktop. Normally this is done through the Administration tools or the Visitor Tools web application. However, these API can be called directly from within the developer's code. For additional information on these API refer to the [Javadoc](#).

Appendix

This appendix contains information on the following subjects:

- [netuix-config.xml](#)
- [Performance](#)

netuix-config.xml

The `netuix-config.xml` file is governed by the XML schema definition file `netuix-config.xsd`. This file contains settings that can be modified to change the behavior of the portal framework. This file is Web application scoped, and the Web application must be redeployed to pick up changes to the file. The file contains the following elements and attributes:

customization

Enable (default) or disable customization. The portal has two modes of operation. In one mode, users and administrators are allowed to customize the portal using a browser (add/delete portlets, pages, books, etc.).

In this mode the portal EJBs and a database must be deployed. In the second mode (the browser hitting the `.portal` file directly) no customization is allowed, so no database or portal EJBs are required. This flag is an indicator to the system what mode of operation the portal is in. To run “light portal” this element must be set to false. To pick up portlets, layouts, shells, look and feels, and themes in the database, this element must be set to true.

pageflow

Enable (default) or disable Page Flow components. In order to run Page Flow portlets in the portal, this element must be enabled. If you are not running Page Flow applications

you can set this element to false and pick up some performance improvements especially during iterative development.

entitlements

Turn entitlement checking off at runtime. The value of the resource-cache-size attribute is a size for the control resources cache. This size depends on the number of desktops, portlets, pages, placeholders, and books contained in a portal. The size can be determined by enabling debug for

`com.bea.netuix.servlets.entitlements.ControlResource` in the `debug.properties` file located in the domain directory. Debug prints out the size for a portal when clicked on all the pages of a portal. The size to be used is from the last line that is printed on the console after clicking on all the pages (all top-level and inner pages).

localization

Enable (default) or disable localization. When localization is enabled, the portal framework attempts to deliver localized content based on a directory search. Set the enable element to "true" to enable localization. Disable localization by setting this element to "false". For performance reasons, if the portal Web application is prepared to deliver localized content, disable localization. To specify a locale provider, include the locale-provider element with value set to the class of the desired LocaleProvider implementation.

default-locale

Default local used thought the system.

propagate-preferences-on-deploy

The propagate-preferences-on-deploy element specifies if portlet preferences should be propagated to the underlying preference store or not. If this element is present, portlet preferences will be propagated to the underlying preference store. If the attribute propagate-to-instances is true, portlet preferences will also be propagated to instances created out of portlets.

reload-database-on-redeploy

Because of the way iterative development redeploys the Web application on any changes to a control or Page Flow, this will stop the database from being reloaded if you are in development mode (i.e. `!AppDescriptor.isProductionModeEnabled`). This defaults to 'false'. If you are in development mode and you want the database to be reloaded from the Web application on a redeploy, set to 'true'.

window-state

The window-state element describes properties of container-supported window states. This element has child elements for each of the different states. You can specify image names and localized alternate text.

window-mode

The window-mode element describes properties of container-supported window modes. This element has child elements for each of the supplied modes, and you can define your own modes.

```
<!-- Example of a custom mode in two languages -->
<window-mode name="SourceViewToggleButton">
  <activate-image>titlebar-button-source.gif</activate-image>
  <deactivate-image>titlebar-button-source-exit.gif</deactivate-image>
  <alt-text>
    <locale language="en">
      <activate>Source View</activate>
      <deactivate>Leave Source View</deactivate>
    </locale>
    <locale language="es">
      <activate>la vista de la fuente</activate>
      <deactivate>Salga la fuente la vista</deactivate>
    </locale>
  </alt-text>
</window-mode>
```

To use the above custom mode in a portlet or book you can do the following:

```
<netuix:titlebar>
  <netuix:modeToggleButton name="SourceViewToggleButton"
    contentUri="/source.jsp" />
  <netuix:minimize/>
</netuix:titlebar>
```

validation

Determines whether XML schema validation is performed on the different XML documents that WebLogic Portal parses.

include-files

Enable (default) or disable validation of '.pinc' files.

dot-files

Enable (default) or disable validation of '.' files (portlet, theme, layout, laf). Turning this off can speed up WebLogic Portal start times and redeployment times. However, turning validation off and allowing invalid files causes error.

control-state-location

The control-state-location element specifies the location for storing control state. All the state for the portal framework can be configured to be stored in different ways; each having advantages and disadvantages. The state includes: current page, active pages, window state (minimize, maximized, etc.). Control state is not the same as application state. Application state is up to each developer.

session

Stores the control state in the user's HTTP session. This is the default control state location. The control state can be preserved until the end of the session.

url

Encodes the control state in portal framework generated links. The control state can be preserved forever. However, clients (browsers) may have limits on the maximum number of characters. When the control state length exceeds the specified maximum number of characters, the portal framework automatically switches the state location to HTTP session.

cookie

Stores the control state as cookies. The expires attributes may be used to specify the lifetime (in seconds) of cookies user for storing control state.

desktop-not-entitled-error-code

The desktop-not-entitled-code element defines the error code to return for access to a desktop denied by an entitlement. The valid options for code are:

401 - Unauthorized

403 - Forbidden (default)

404 - Not found - use if you don't want to let the user know the resource exists.

Performance

Running Weblogic Portal Server in Production SETUP

1. To start portal server, use this command “startWebLogic.cmd/startWebLogic.sh nodebug production notestconsole noiteratedev noLogErrorsToConsole nopointbase”.
2. In workshopLogCfg.xml and workshopLogCfgVerbose.xml files change priority value to “warn” for all the categories. These files contain configuration information for logging by log4j framework. These files can be found under
<install-dir>/bea/weblogic812/common/lib.

3. Deploy all the portal web-applications with `servletReloadCheckSecs` set to “-1” in `config.xml`.

Configuration in netuix-config.xml

The `netuix-config.xml` contains portal framework related configuration information. It is web-application scoped.

- The “customization” element is a switch to indicate if a portal is customizable or not. If a portal is served from a `.portal` file (rather than from a database) and users are not allowed to customize it then customization could be disabled by setting “enable” element’s value to “false”. If a portal supports customizations then customization should be enabled.
- The `pageflow` element is a switch to enable or disable pageflows usage in a portal, disable it if a portal is not using any pageflows.
- The “validation” element is a switch for validating portal related files such as
- `.pinc`, `.portlet`, and `.portal` files. Disable validation when running portal server in production setup.
- The “entitlements” element is a switch to indicate that a portal is setup to use entitlement policies (users to portal resources such as desktop, books, pages, portlets, etc). Disable entitlements if a portal is not using any security policies. If a portal is using security policies enable it and set the value for “control-resource-cache-size” attribute using “control-resource-cache-size” = num of desktops + num of books + num of pages + num of portlets + num of buttons (max, min, help, edit) used in a portal. The default value could be used if memory is a concern.

Note: By default, the entitlements switch is off in the `netuix-config.xml` file. See the [Performance Tuning Guide](#) for instructions on how to enable entitlements.

- The “localization” element is a switch to indicate that a portal supports multiple locales. This could be disabled if a portal supports only one locale.

Cache Configuration

The `application-config.xml` file contains settings for all the caches, it can be found under `<enterprise-application-dir>/META-INF/`. For a database based (streaming) portal, the max entries for “portalControlTreeCache” cache should be set to a value based on num of users, available memory and portal size. An ideal value is equal to number of users plus one. If a portal has floatable portlets the above rule should be applied in finding an optimal value for max entries for “portletControlTreeCache” cache.

Note: Don't change "TimeToLive"; it is set to "-1" by default.

Static Files

The look and feel of a portal uses `css`, `js` and `gif` files. The performance and scalability will improve if these static files are served from a different web server.

Portal Size

The performance and scalability of a portal application also depends on a portal size, number of books, number of pages, number of portlets and number of buttons. The time taken to serve a portal from file or database depends on portal's size as it involves XML parsing.

A portal served from database is cached (`portalControlTreeCache`) to avoid going to the portal database from second time onwards. The portal is cached for each user, if users have customized their portals. The memory usage goes up as the number of users with customizations increase. Download the [portal_size.jar](#) tool to find out memory and response statistics for a portal.

Notes: The memory size calculated is not accurate.

The tool calculates the size of the full control tree, creating statistics for portals whether or not `treeOptimization` is active.

Portlet Content

The portlet's content is referenced by `contentUri` element in `.portlet` file, if it is expensive to compute content every time, consider using `renderCacheable` portlet attributes to cache portlet's content or if the portlet's content type is `jsp`, consider using `wl:cache jsp` tags to cache static portions within a `jsp`.

JVM Parameter Values

The following JVM parameters shown in the following tables:

Hardware Configurations

JVM	W2K	Linux	Solaris
BEA	-Xms1024	Xms1024	-Xms1024
jrocket81sp2_141_05	-Xmx1024	-Xmx1024	-Xmx1024
	-Xgc:parallel	-Xgc:parallel	-Xgc:parallel
JDK 1.4.1_05	-Xms1024	-Xms1024	-Xms1024
server VM	-Xmx1024	-Xmx1024	-Xmx1024
	-XX:NewRatio=2	-XX:NewRatio=2	-XX:NewRatio=2
	-XX:MaxPermSize=128m	-XX:+UseParallelGC	-XX:+UseParallelGC
		-XX:MaxPermSize=128m	-XX:MaxPermSize=128m

High Performing JVM parameters

Platform Name	CPUs * CPU speed	CPU type	Physical Memory	CPU bits	Hardware Model	OS
W2K	1 * 3.0GHz	Intel Pentium® 4	2048MB	32	Dell 650	Microsoft Windows 2000, Advanced Server
Linux	1 * 3.0GHz	Intel Pentium® 4	2048MB	32	Dell 650	Red Hat Linux Advanced Server release 2.1AS/i686 (Pensacola)
Solaris	2 * 1002MHz	Sun's sparcv9 processor	2048MB	64	Sun 240V	SunOS 5.9

Appendix