Dissecting a Page Template

István Kiss
WebCenter Team, Oracle Corporation
March 2013

Contents

Overview

In this document we analyze the details of a page template for an Oracle WebCenter Portal Framework application. Although this is a relatively simple page template, but we hope that if you follow the details of the code, you will learn how to build a similar, or even more complex, page template. We cover a number of recommended practices for creating a page template.

This document is not a complete description of page template creation. It does not try to replace the documentation and it will not show the detailed steps of how to work with JDeveloper to create such a template.

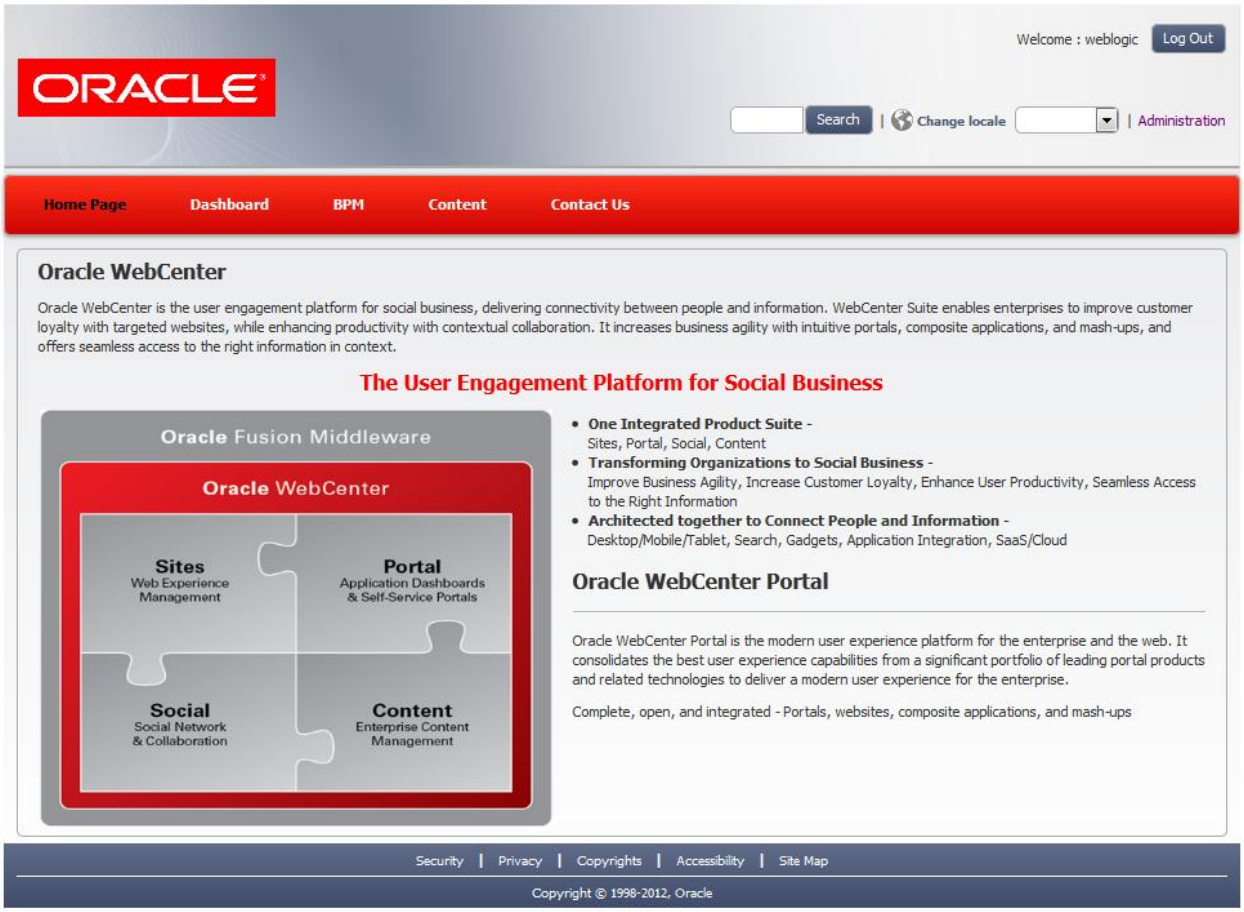
In order to fully understand this document, we assume that:

- You are familiar with Oracle JDeveloper, you have created and run at least a simple, "out-of-the-box" WebCenter Portal Framework application.
- You have some experience with ADF user interface components and CSS style sheets.
- You have followed the online training module: *Creating and Using Page Templates in Oracle WebCenter Portal Applications* that is published on the Oracle Technology Network - Oracle WebCenter Portal Online Training page.

To learn more about WebCenter Portal Framework applications in general and their page templates, refer to the list in Related Documents, published with the training module.

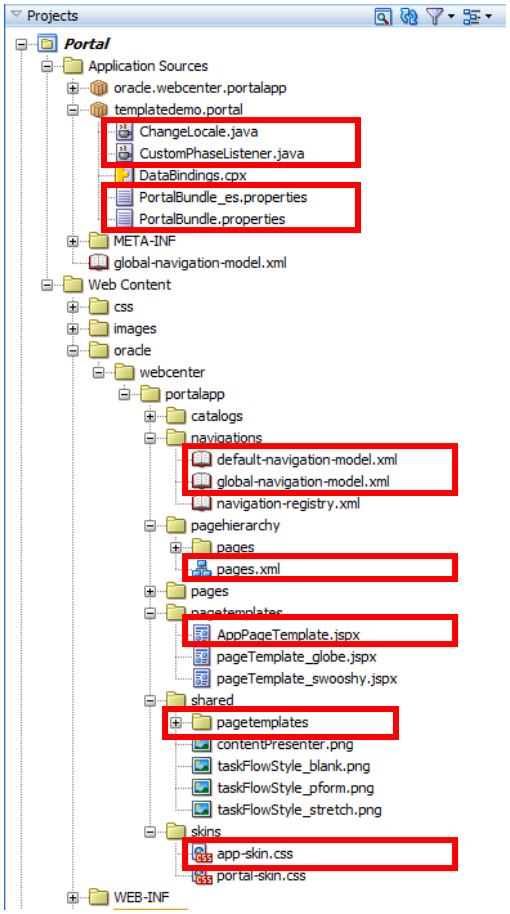
The example application was created with Oracle WebCenter Portal 11.1.1.6.0. If you want to test our code, you need to download JDeveloper 11.1.1.6.0 and install the WebCenter Portal extension.

For the initial version of the page template to be dissected, credit goes to Martin Deh from the Oracle Fusion Middleware A-Team. Here is the home page of the application, as it is displayed to an authenticated user with administrative privileges. Parts of the page: the header, the navigation bar, and the footer are rendered by the page template; the center part, the one with a thin border, is the content of the application's home page.



Application Files

Although the page template is a single file, the working page template relies on components described in other files. Here is a partial list of the files that we use with our templates or in the demo.



1. `AppPageTemplate.jspx` is the application template.
2. Our template renders two navigation models:
   - `default-navigation-model.xml` contains links to the application pages.
   - `global-navigation-model.xml` contains the global links rendered in the page footer.
3. `pages.xml` contains the page hierarchy, which is used to define the page access security model.
4. `app-skin.xml` is the application-specific skin file that extends `portal-skin.css` and contains CSS style definitions used in the page template and in application pages.
5. `ChangeLocal.java` and `CustomPhaseListener.java` files are necessary to implement the application internationalization.
6. The two properties files store the texts visible on the page template and on other pages. `PortalBundle.properties` holds the texts in the default language (English), while `PortalBundle_es.properties` is the Spanish translation of all the texts from `PortalBundle.properties`.
7. The `/oracle/webcenter/portalapp/shared/pagetemplates` folder contains images used in the page template.

Let's see some of these files in details. The page template itself, `AppPageTemplate.jspx`, will be analyzed later in the document.

Resource Bundles

Resource bundles are special Java components that support internationalization of the application. Resource bundles can be Java classes, but here we use simpler, property file-based resource bundles.

Each of the property files contains a list of key-value pairs, where the key is used to look up a text that will be displayed on the page. For example, in the default resource bundle (`PortalBundle.properties`) you can see the following key-value pairs, and many more:

```
home_title=Home Page
login_title=Login Page
welcome=Welcome
admin=Administration
back=Back to Portal
logout=Log Out
username=Username
password=Password
login=Login
```

To internationalize your application you have to create a translation of the texts and store it in another resource bundle file, for example `PortalBundle_es.properties`, which holds the Spanish translations:

```
home_title=Página de inicio
login_title=La página de acceso
welcome=Bienvenida
admin=Administrador
back=Volver al Portal
logout=Finalizar la sesión
username=Nombre de usuario
password=Contraseña
login=Iniciar sesión
```

In our page template and in the application pages all of the texts displayed come from the current resource bundle file, using a key to look up the current text. For example the following component:

```
<af:commandButton …text="#{portalBundle.login}">
```

displays a button where we use the key `login` to look up the actual text from the current resource bundle. So the text displayed is either *Login* or *Iniciar session*, depending on the current locale.

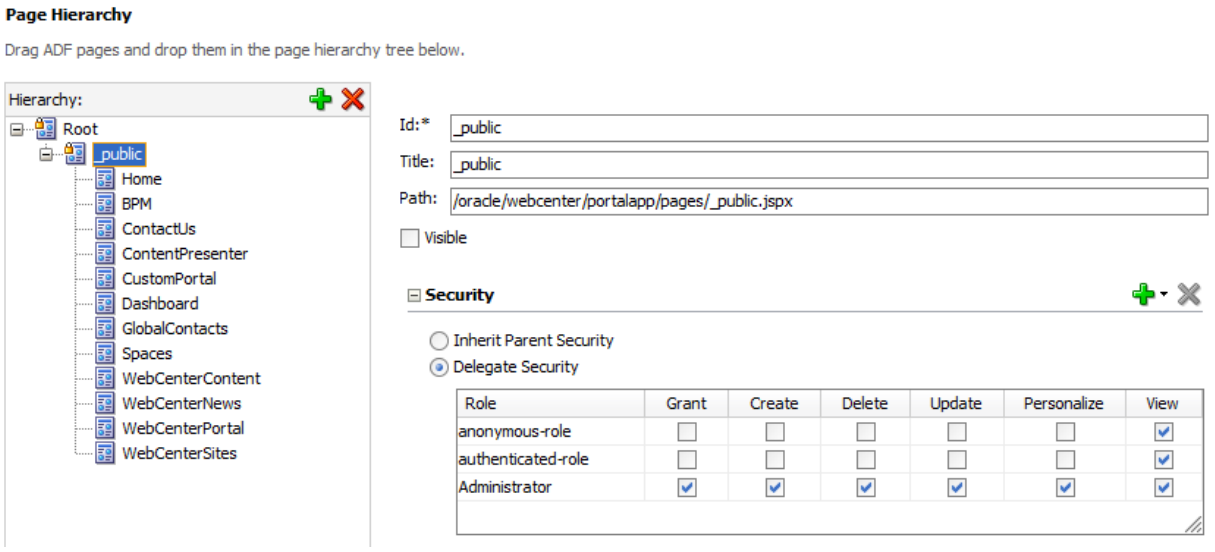
Later on you will see that the user can dynamically change the current Locale, which in turn changes the resource bundle.

Page Hierarchy

In a WebCenter Portal Framework application you can create a tree-like hierarchy of pages. Although this hierarchy can be used to access the pages, we recommend that you use a separate navigation model to define how pages and other application resources can be accessed. Later you will see our navigation models.

The important role of the page hierarchy is to define security, specifically access privileges to the pages. Although you could use ADF security constructs to define page access rights, we recommend that you use the WebCenter Portal-specific page hierarchy, since it provides a centralized administration of these privileges and it is also supported by our run-time administration tool, the WebCenter Portal Administration Console. As a consequence, all of the application pages must be added to the page hierarchy!

In the hierarchy you can set access rights at every level, and those rights are inherited down in the hierarchy, unless you override them. The details of configuring security are beyond the scope of this document. The demonstration application's security model is very simple: there are only a few application pages and all of them have the same security. To set the common security requirements, we created an empty page named `_public`. We defined the security requirements for this page and made all of our application pages a sub-page of `_pubic`. Therefore these pages inherit the security settings.



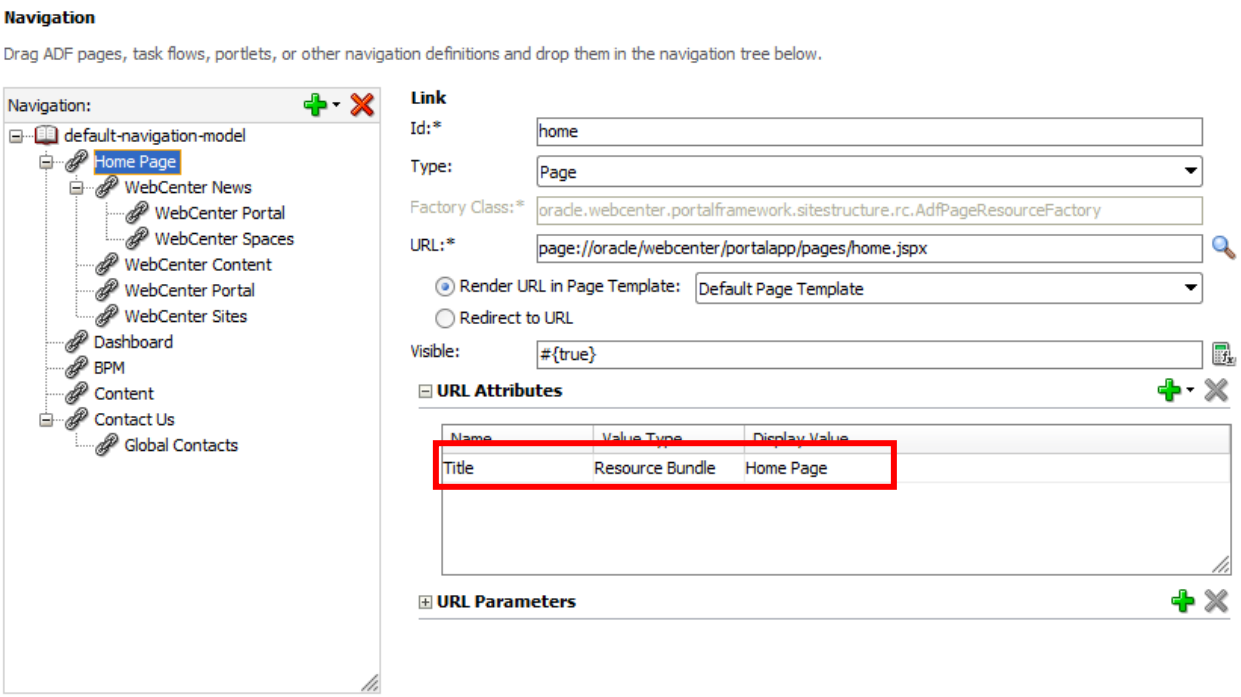
In a more complex application we recommend that you create several empty pages with different security settings and place the application pages as sub-pages of these, depending on the security requirements.

Navigation Models

In our application we use two navigation models: `default-navigation-model.xml` is used to access the application's pages, whereas `global-navigation-model.xml` contains links that are rendered in the page footer.

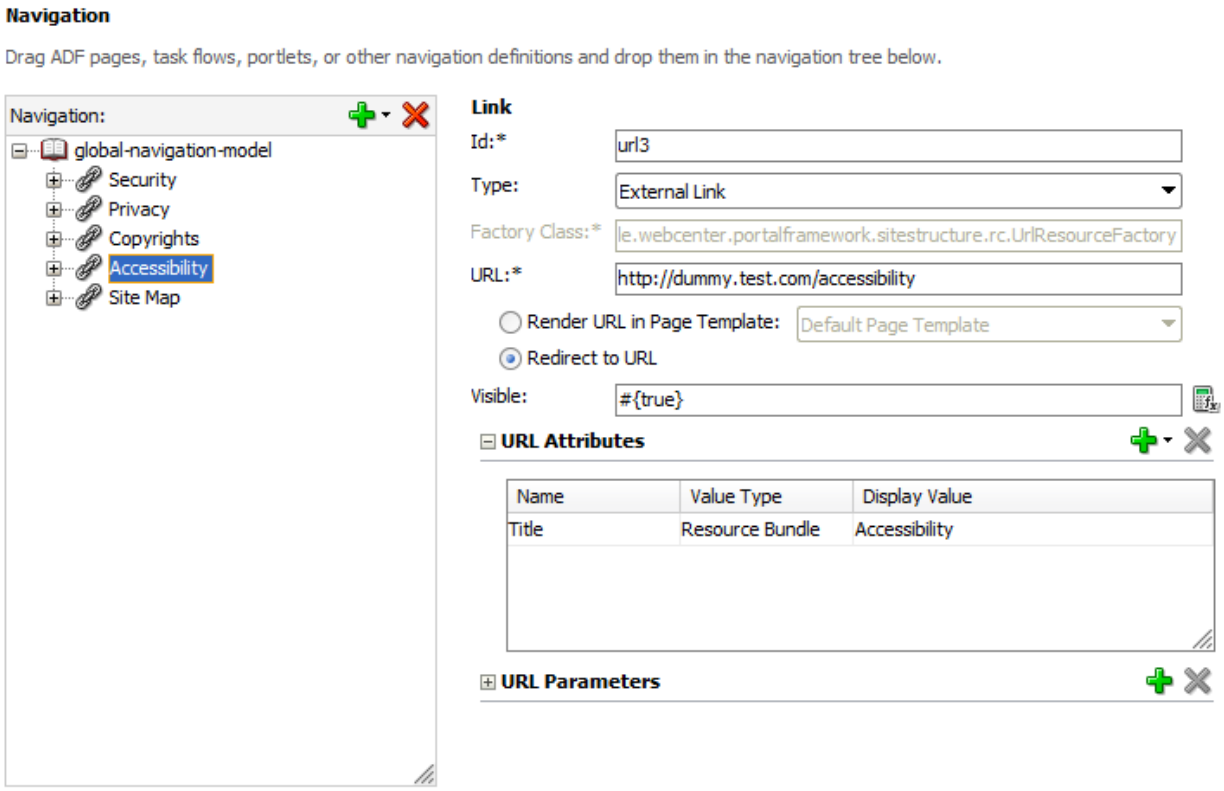
A navigation model can describe a multi-level structure of components that contain not only application pages, but other elements, like external links, content stored in the content server, task flows, portlets, and more.

Here is the graphical representation (Design View) of the default navigation model:



In our example application we use a simple model that contains only links to the application pages, with 3 levels in the structure. In every page link we set the *Title* attribute which will be displayed in the navigation bar. Note that to support internationalization of the application, this attribute is not hardcoded English text, but set in the resource bundle.

The global navigation model is a simple list of components. In our example these are external links, but in a real application these could be links to existing application pages or external pages. Note that the *Title* attribute is also defined using the resource bundle. Below is the visual representation of the global navigation model.

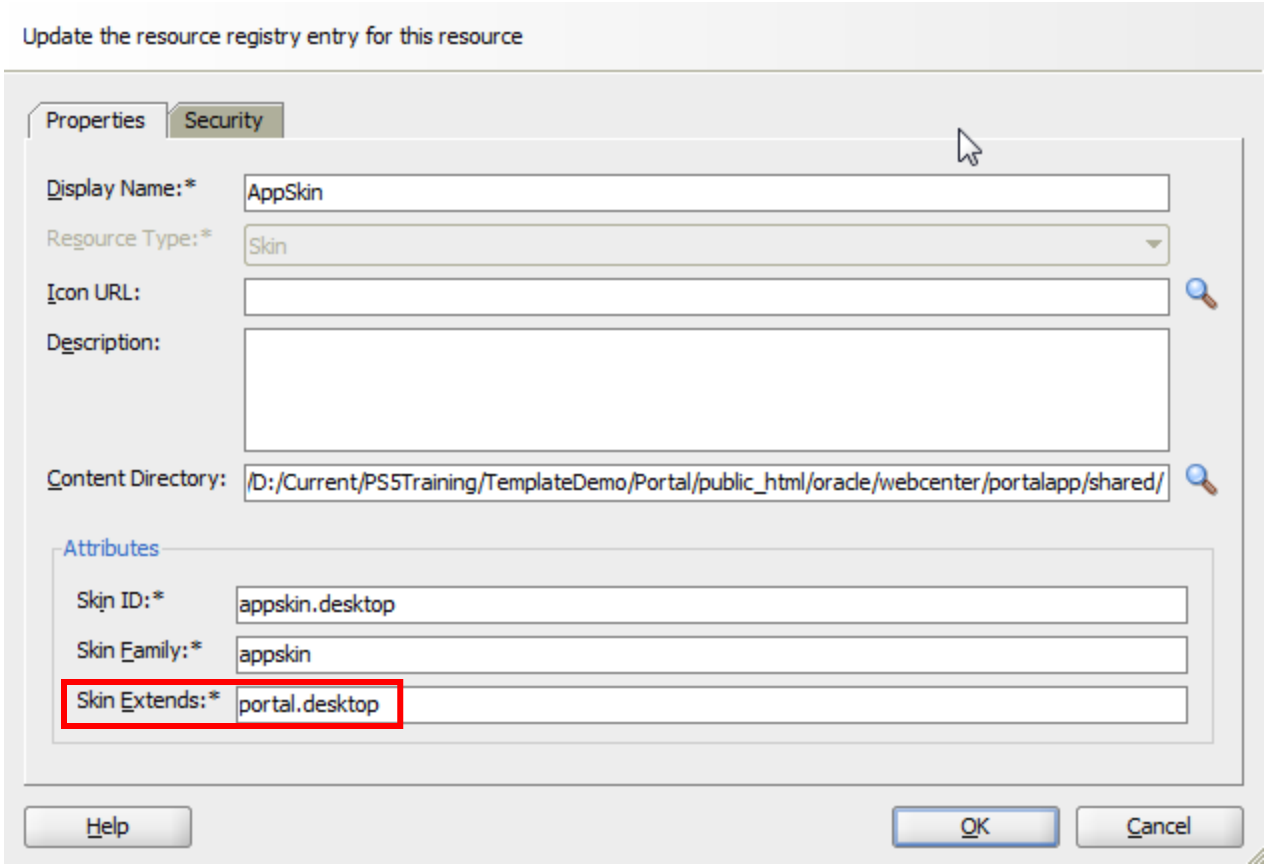


Application Skin

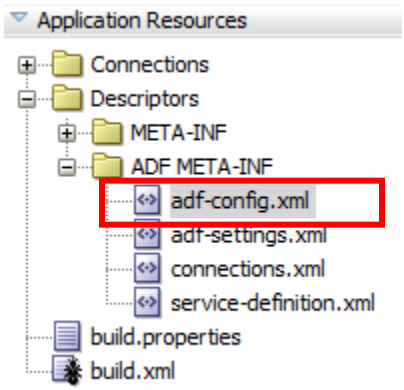
An ADF application's appearance can be customized using skin files. A skin file is an ADF-specific file that contains a set style definitions, similar to CSS (Cascading Style Sheet) definitions. These definitions control how ADF and other user interface elements are displayed by the browser.

WebCenter Portal Framework applications have a default skin file: `portal-skin.css`. Although you can add your CSS definitions to this file, we recommend that you create a new skin file, which extends the default skin. Extending the default skin is simple:

1. Create a new CSS file in the `/oracle/webcenter/portalapp/skins` folder where the default skin is also placed.
2. Create a portal resource from this skin file. Define your skin's Display Name, ID, and Family, and make sure that it extends the default skin `portal.desktop`.



3. Change the application's default skin. Edit `adf-config.xml` file, locate the preference for the skin, and change its value to your skin family.




```
<portal:preference
        id="oracle.webcenter.portalapp.skin"
        desc="Default Portal Skin" value="appskin"
        resourceType="Skin" display="true"/>
```

The details of skinning an ADF application are not in the scope of this document, but here are a couple examples of how skinning is used in the example page template.

You can override existing ADF styles. For example:

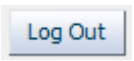
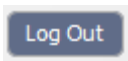
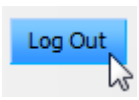
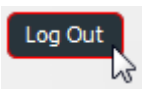
```
af|document {
  background-image: none;
```

```
  background-color: white;
}
```

This example overrides the style for `<af:document>` component, which is the root node of all of the pages. This setting replaces the default gradient background of a page with a simple white background.

You can override default values used in multiple style definitions. For example these settings define how buttons are rendered.

```
.AFButtonBackground:alias {
    background-image:none;
    background-color: rgb(96, 108, 136);
}
.AFButtonBackgroundHover:alias {
    background-image: none;
    background-color: rgb(40, 52, 59);
}
.AFButtonBackgroundActive:alias {
    background-image:none;
    background-color: rgb(96, 108, 136);
}
.AFButtonBackgroundFocus:alias {
    background-image:none;
    background-color: rgb(96, 108, 136);
}
.AFButtonBorder:alias {
    border: solid #ABABAB 1px;
    border-radius: 5px;
}
.AFButtonBorderHover:alias {
    border:solid red 1px;
}
.AFButtonForeground:alias {
    text-align: center;
    white-space: nowrap;
    color: white;
}
.AFButtonForegroundHover:alias {
    color: white;
}
```

	Default skin	Custom skin
Normal		
Hover over		

You can also define your styles that can be used only with specific ADF components.

```
af|image.sidebarimg {
```

```
    width:160px;
}
```

For example this style is used only in an `image` component.

```
<af:image … styleClass="sidebarimg"/>
```

You can define your own styles:

```
.footerText {
    color: white;
    font-size: x-small;
    text-decoration:none;
}
```

These styles can be used with any ADF component. For example:

```
<af:goLink … styleClass="footerText">
```

Styles you define can be applied to non-ADF components as well. For example the style `navbar`

```
.navbar {
    z-index: 130;
    height: 35px;
    margin: 5px 0px;
    padding: 5px;
    line-height: 100%;
    background: #ff3019;
    border-radius: 5px;
    border: solid #ABABAB 1px;
    box-shadow: 0 1px 3px rgba(0, 0, 0, .4);
}
```

will later be applied to an HTML `<ul>` tag:

```
<ul id="nav" class="navbar">
```

You can also define styles used for specific HTML tags below a node in the document hierarchy (DOM tree) with a given style.

```
.navbar a {
    font-weight: bold;
    color: white;
    text-decoration: none;
    display: block;
    padding:  6px 20px;
    border-radius: 5px;
}

.navbar a.currentNode {
  color: black !important;
}
```

The first applies to any anchor tag (`<a …>`) under a node with the `navbar` style, while the second applies only to those anchor tag, where we defined a style class called `currentNode`.

```
<ul id="nav" class="navbar">
 …
  <a href="…" …> … </a>
  <a class="currentNode" href="…" …> … </a>
 …
</ul>
```

Some of the CSS features are not recognized the same way by all of the browsers. For example to create a gradient background for an HTML component, different browsers require different syntax. Mozilla Firefox uses the following syntax:

```
-moz-linear-gradient(top, #ff3019 0%, #cf0404 100%)
```

while Google Chrome takes another syntax:

```
-webkit-gradient(linear, center top, center bottom,
                 color-stop(0%, #ff3019), color-stop(100%, #cf0404))
```

In order to render correctly in both browsers, we have to include both settings in our style definition. ADF skin file syntax provides a conditional evaluation of CSS settings. For example in the style definition we provide a default uniform background color for those browsers that cannot render a gradient background.

```
.navbar {
    …
    background: #ff3019;
    …
}
```

In the skin file we also provide settings that will override the default in case the actual browser is Firefox (`@agent mozilla`) or Chrome (`@agent webkit`). Similarly you can also add support for other browsers.

```
@agent mozilla {
  …
  .navbar {
      background: -moz-linear-gradient(top, #ff3019 0%, #cf0404 100%);
  }
  …
}

@agent webkit {
  …
  .navbar {
      background: -webkit-gradient(linear, center top, center bottom,
                   color-stop(0%, #ff3019), color-stop(100%, #cf0404));
  }
  …
}
```

Note that in the conditional style definitions we override the background attribute only. Other attributes are inherited from the default style.

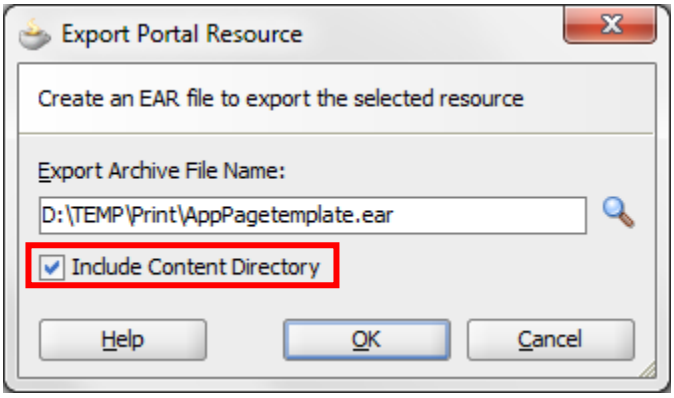
Shared Files

A page template and other portal resources might require additional files. Our example uses icons and logo images. A more complex example might use JavaScript libraries, additional CSS files, and even some static HTML files.

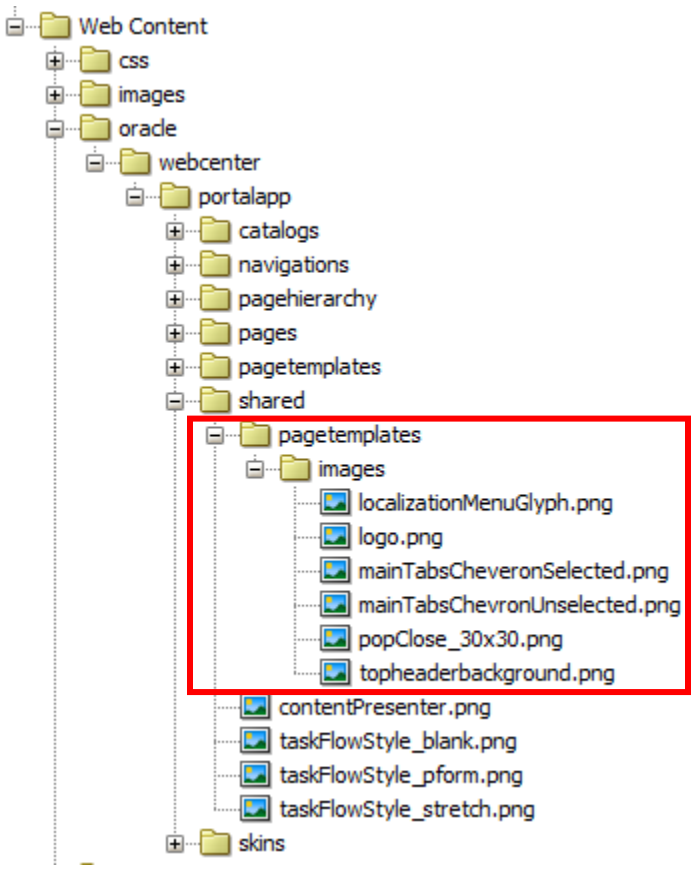
These files have to be deployed in the application server and made accessible to your application. The easiest way to add them to your application is to place them somewhere under the Web root, i.e. under the folder, which is accessible through a Web request.

As you know, a page template is a portal resource. Portal resources are packaged into special archives. These archives can be imported at design- and run-time into WebCenter Portal Framework (and WebCenter Portal Spaces) applications. It is an obvious requirement that all other files required by this resource should be packaged together in the same archive.

Beginning with WebCenter 11.1.1.6.0, when a resource archive is created, you can add the content of the `/oracle/webcenter/portalapp/shared` folder into the archive. See the checkbox below.



We wanted to separate the page template-related resources from the others, so we created a special subfolder, `/oracle/webcenter/portalapp/shared/pagetemplates/images` that includes all of the images used in the template.



Page Template

The page template is a JSPX file. Since the page template is a portal resource, it should reside under the `/oracle/webcenter/portalapp` folder. We recommend that you create the application's page templates in the same folder as the out-of-the-box templates: `/oracle/webcenter/portalapp/pagetemplates`.

Structure of the Template File

This is the overall structure of the page template JSPX file.

```
<?xml version='1.0' encoding='UTF-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
          xmlns:pe="http://xmlns.oracle.com/adf/pageeditor"
          xmlns:cust="http://xmlns.oracle.com/adf/faces/customizable"
          xmlns:c="http://java.sun.com/jsp/jstl/core">

   ... Next comes the variables we define ...

  <jsp:directive.page contentType="text/html;charset=UTF-8"/>
  <af:pageTemplateDef var="attrs">
   ... Next comes the page template ...
```

```
    <af:xmlContent>
      <component xmlns="http://xmlns.oracle.com/adf/faces/rich/component">
        <display-name>Application Page Template</display-name>
        <facet>
          <description>Facet for content</description>
          <facet-name>content</facet-name>
        </facet>
```

… Finally the page template attributes …

```
      </component>
    </af:xmlContent>
  </af:pageTemplateDef>
</jsp:root>
```

Note that we will use the core of JSTL (JavaServer Pages Standard Tag Library) tags, like `<c:set …>`, `<c:forEach …>`, and `<c:if …>` tags in the page template, therefore there is an additional XML namespace defined, as indicated with blue highlight in the `<jsp:root>` tag.

Note the `display-name` and `facet` tags highlighted in yellow above. Display name is used when the template is displayed, for example in JDeveloper's New Page wizard, where you can choose which of the templates to use for the new page.

Facets are placeholders in the template which will be filled in by the pages using the template. An ADF page template can have any number of facets, but a WebCenter Framework application requires that each template have a facet called **`content`**. Typically we do not use any other facets, since Portal's run-time tools recognize only this facet. Other facets could be used only at design time, in JDeveloper.

Attributes and Variables

Although the page template attributes are typically at the end of the template file, we discuss their roles first.

These attributes are similar to method parameters: their values can be set in the page where the template is used, but they can also have default values, in case the attribute is not set by the page. For example in `login.jspx` two of these attributes are set:

```
<af:document title="#{portalResource['login_title']}" id="d1">
  <af:pageTemplate
      value="#{bindings.pageTemplateBinding.templateModel}" id="pt1">
    <f:attribute name="showNavigation" value="#{false}"/>
    <f:attribute name="showLogin" value="#{false}"/>
  …
</af:document>
```

Other attributes will have the default value.

Inside the page template you can refer to these attributes in EL expressions, for example

```
#{attrs.contentWidth}
```

You do not have to define page template attributes - a page template without attributes is syntactically correct. However, page template attributes are useful when you want to use one page template for several pages where the template should render slightly differently. This can be controlled by the page template attributes' values. The following attributes are copied from an out-of-the-box page template. Some of them are used to show or hide components of the template.

```
<attribute>
  <attribute-name>contentWidth</attribute-name>
  <attribute-class>java.lang.String</attribute-class>
  <default-value>960px</default-value>
</attribute>
```

The `contentWidth` attribute sets the page width. Default value is `960` pixels.

```
<attribute>
  <attribute-name>showNavigation</attribute-name>
  <attribute-class>java.lang.Boolean</attribute-class>
  <default-value>#{true}</default-value>
</attribute>
<attribute>
  <attribute-name>showLogin</attribute-name>
  <attribute-class>java.lang.Boolean</attribute-class>
  <default-value>#{true}</default-value>
</attribute>
```

`showNavigation` and `showLogin` attributes can be used to show or hide the login link and the navigation bar. For example `login.jspx` may use the page template, but in this page we must not display any of these elements, so their values are set to false, as you can see in the example above.

In our page template we do not use the separate login page, `login.jspx`. Login is done using a pop-up window, so these attributes essentially always remain true. Still, in order to show how to create a general purpose page template, we still use the attributes in the template to show or hide these components.

```
<attribute>
  <attribute-name>showGreetings</attribute-name>
  <attribute-class>java.lang.Boolean</attribute-class>
  <default-value>#{securityContext.authenticated}</default-value>
</attribute>
```

`showGreetings` controls, if the greeting message is displayed on the page, this case for authenticated users only (`#{securityContext.authenticated}`).

```
<attribute>
  <attribute-name>showAdmin</attribute-name>
  <attribute-class>java.lang.Boolean</attribute-class>
  <default-value>#{WCSecurityContext.userInAppRole['Administrator']}
  </default-value>
</attribute>
```

The `showAdmin` attribute controls whether or not the link to WebCenter Portal Administration Console is displayed on the page. Note: here we changed the value found in the out-of-the-box template. In our template, the default value expression

```
#{WCSecurityContext.userInAppRole['Administrator']}
```

evaluates to true if the current user is a member of the *Administrator* application role. The predefined user *weblogic* is a member, but you can add any of your application users or enterprise roles to the Administrator role.

```
        <attribute>
          <attribute-name>isEditingTemplate</attribute-name>
          <attribute-class>java.lang.Boolean</attribute-class>
          <default-value>#{false}</default-value>
        </attribute>
      </component>
    </af:xmlContent>
```

Finally the `isEditingTemplate` attribute indicates if the page template is being edited. Using the Administrator Console it is possible to edit at run time either a page template or any of the application pages. This attribute is set at run time, by the Composer. When a page is being edited, we do not want to show the template but only the content of the page. When the page template is edited, we should show the template's elements.

In addition to the attributes, we can define variables. These variables are set when the template starts rendering and we can use their values inside the template.

```
    <c:set var="portalBundle"
           value="#{adfBundle['templatedemo.portal.PortalBundle']}"/>
```

The `portalBundle` variable refers to the actual resource bundle. The default bundle name is set here, but at run time the actual bundle depends on the current locale.

```
    <c:set var="contextRoot"
           value="${facesContext.externalContext.requestContextPath}"
           scope="session"/>
```

This is the context root of the current application. Rather than hard coding the root, it is dynamically evaluated using the URL from the current request.

```
    <c:set var="images"
           value="/${contextRoot}/oracle/webcenter/portalapp/shared/
  pagetemplates/images"
           scope="session"/>
```

All of the images, for example the logo image, or the search icon is placed under the application's shared folder. The `images` variable contains URL to this folder.

```
    <c:set var="showTemplate"
           value="${!composerContext.inEditMode or attrs.isEditingTemplate}"
```

```
        scope="session"/>
```

As mentioned earlier, there are times when elements of the page template have to be rendered or hidden. The variable evaluates to true when the application is not in edit mode or we edit the template itself.

Creating the Layout

Our page template implements a flow layout. Flow layout is used by the majority of web sites. The characteristics of the flow layout are:

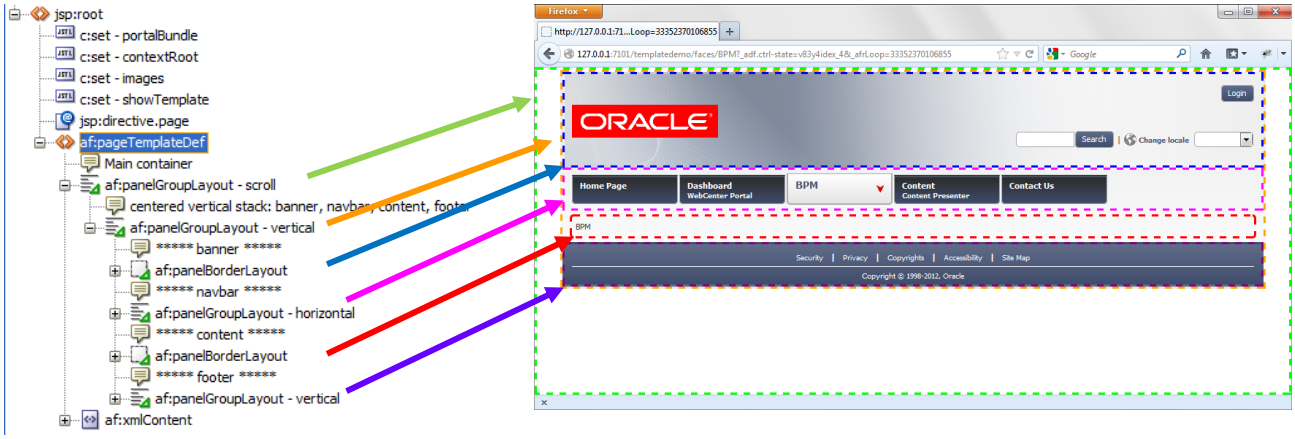
- Components have fixed size and they are arranged side-by-side or one below the other.
- The page has a fixed width. The combined widths of the components never exceed the page width. Some components might be stretched vertically to fill up the available space.
- If the browser window is narrower than the page width, the browser displays a horizontal scroll bar.
- The page height is defined by the combined heights of the components.
- If the page is higher than the browser window, the browser displays a vertical scroll bar.
- As a consequence, the top (header) and/or the bottom (footer) of the page might not always be visible.

Our page has a simple vertically stacked layout. The top of the page contains a common page header, followed by a navigation area. Below the navigation the page content is displayed. Finally the bottom of the page contains a common footer.

1. The lime colored root container is a scrollable `panelGroupLayout` that fills the browser area and creates the browser scroll bars, if necessary.
2. The orange main `panelGroupLayout` container holds all of the user interface components. Additional CSS style allows this container to flow in the center of the main container.

Inside this container, stacked vertically:

3. The blue `panelBorderLayout` container holds the header elements.
4. The fuchsia `panelBorderlayout` container holds the navigation buttons.
5. The red `panelBorderLayout` container contains the page content.
6. Finally the purple `panelGroupLayout` holds the components of the page footer.



Here are the container components of the page template that define the general layout of the page:

```
<af:panelGroupLayout id="pt_root" layout="scroll"
                     inlineStyle="margin: 5px 0;">
```

This is the root, the outermost container, which holds all the components of the page template. Since we are creating a flow layout, it is recommended to use a scroll-type `panelGroupLayout` as the outermost component. This container covers the browser window, but it does not stretch its content, which are the components inside. As a result, it provides vertical and horizontal scroll bars if the content of this container does not fit into the available space.

We also added a 5 pixel margin to the top and bottom of the page, defining directly the style of this component (`inlineStyle="margin: 5px 0;"`).

```
<!-- centered vertical stack: banner, navbar, content, footer -->
<af:panelGroupLayout id="pt_main"
   layout="vertical" styleClass="mainStyle"
   inlineStyle="width:#{attrs.contentWidth}; margin: 0 auto;">
```

Inside the root container we use a container that holds the user interface elements. Note that we must use only such containers that cannot be stretched and do not stretch their content, for example a `panelGroupLayout` container with vertical layout style. Vertical layout arranges the following one below the other.

In our flow layout design, every page has a fixed width and if the browser window is wider than this width, we want the page content to float in the center of the browser window. Note the `inlineStyle` attribute sets the width of the content and the margin setting centers the content.

```
inlineStyle="width:#{attrs.contentWidth}; margin: 0 auto;"
```

Also note that the container has a dedicated style class, `mainStyle`, defined in the application skin. The skin defines a background color for the whole page. The default uniform color will be overridden by a gradient color, if the browser supports it.

```
.mainStyle {
```

```
    background: #E6E7E9;
}
```

The following four containers hold the 4 regions of the page: the top banner, navigation bar, page content, and footer.

```
<!-- ***** banner ***** -->
<af:panelBorderLayout id="pt_banner"
                      styleClass="bannerStyle AFStretchWidth"
                      rendered="#{showTemplate}">
  ... Here is the banner ...
</af:panelBorderLayout>
<!-- ***** navbar ***** -->
<af:panelGroupLayout id="pt_navbar" layout="horizontal"
                     styleClass="navbarStyle AFStretchWidth"
                     rendered="#{showTemplate}">
  ... the navigation bar ...
</af:panelGroupLayout>
<!-- ***** content ***** -->
<af:panelBorderLayout id="pt_content" styleClass="contentStyle">
  <!-- content facet -->
  <af:facetRef facetName="content"/>
</af:panelBorderLayout>
<!-- ***** footer ***** -->
<af:panelGroupLayout id="pt_footer" layout="vertical" halign="center"
                     styleClass="footerStyle AFStretchWidth"
                     rendered="#{showTemplate}">
  ... finally the page footer ...
</af:panelGroupLayout>
```

The top banner and the content regions are implemented with `panelBorderlayout`, while the navigation bar and footer use a simpler container: `panelGroupLayout`. Both container types can be used in flow layout, for they are not being stretched and do not stretch their children.

You can see that three out of the four containers (banner, navigation bar, and footer) contain elements of the page template. These containers are rendered only when the page template has to be rendered and are controlled by their attribute: `rendered="#{showTemplate}"`. The fourth one, the content container, contains the page content and always has to be rendered.

Note that the `styleClass="bannerStyle AFStretchWidth"` attribute ensures that the container stretches horizontally so it fills the width of the page, which is defined in the surrounding `panelGroupLayout` element.

As you can see all of these containers each have their own style class (`bannerStyle`, `navbarStyle`, `contentStyle`, `footerStyle`) that controls their appearance.

```
.bannerStyle {
    padding: 0 10px;
    border: 1px solid #E6E7E9;
    border-top: none;
    border-bottom: 2px solid #ABABAB;
    background-image:
```

```
url('/oracle/webcenter/portalapp/shared/pagetemplates/images/topheaderb
ackground.png');
}

.navbarStyle {}

.contentStyle {
    margin: 5px 10px;
    border-radius: 5px;
    border: solid #ABABAB 1.5px;
}

.footerStyle {
    padding: 5px 10px;
    background: rgb(96, 108, 136);
}
```

Currently `navbarStyle` is empty; we define the general style of this area later, when rendering the navigation bar. All the other styles provide some padding for their content. Please note that most of them use 10 pixels of padding on the left and right side (`padding: 5px 10px;`), so their content's edges are aligned nicely horizontally.

Note, that the banner has its own background image (`topheaderbackground.png`) which is stored in the shared folder. The footer has a colored background. The other two inherit their background from the main container.

What remains from the layout are the closings of the two outermost containers, the main and the root.

```
    </af:panelGroupLayout>
</af:panelGroupLayout>
```

Page Header Banner

The page header is displayed slightly differently for the public user

than for an authenticated user. For example the default user, *weblogic*, who has administrator privileges in the WebCenter Framework application:

You can see that there are common elements, like the logo image, the search field, and the change locale drop-down list.

While the unauthenticated user sees a Login button, the authenticated user sees a greeting message with the user's name and a logout button instead. If the user has administrator privileges, a special Administration link is also rendered in the banner.

The following diagram illustrates the detailed layout of the banner.


```
<af:panelBorderLayout id="pt_banner" styleClass="bannerStyle
AFStretchWidth"
                      rendered="#{showTemplate}">
```

As we have seen earlier, the header banner is rendered by a horizontal `panelBorderLayout` container. It has its own style and it is rendered only when the template has to be displayed.

The `panelBorderLayout` is a powerful container that can have several facets like top, bottom, start, and end that are arranged around the container's content, providing an easy way to build table-like layouts. Here we use two facets of the container: the start facet displays the logo image and the end facet contains the rest of the components.

```
<!-- Logo image -->
<f:facet name="start">
    <af:goLink id="home" destination="/">
      <af:image source="#{images}/logo.png"/>
    </af:goLink>
</f:facet>
```

Clicking the logo image should take the browser to the home page of the application.

Note that the logo image is stored in the shared folder for the page template images (`source="#{images}/logo.png"`). The logo image is displayed inside a `goLink` tag which take us to the context root of the application (`destination="/"`).

Note: According the `web.xml` file, the context root request returns the `index.html` file, which is the welcome file of the application.

```
<welcome-file-list>
  <welcome-file>/index.html</welcome-file>
</welcome-file-list>
```

The `index.html` file redirects the browser to `<context root>/faces/wcnav_defaultSelection`

```
<meta http-equiv="refresh"
content="0;url=./faces/wcnav_defaultSelection" />
```

This URL returns the default page from the default navigation model, which is our home page.

The end facet of the banner container holds all the other elements of the banner, like the login/logout links, greetings, search and locale changer.

```
<f:facet name="end">
  <af:panelGroupLayout layout="vertical" id="pt_pg12" halign="end"
                       inlineStyle="padding: 15px 0px">
```

The end facet holds a vertical style `panelGroupLayout` container. The components in this container are right-aligned (`halign="end"`) and the container has a 15 pixel top and bottom padding (`inlineStyle="padding: 15px 0px"`).

This container has 3 rows, i.e. 3 groups of components, aligned on top of each other. The first row holds the greeting message and the login/logout link. The second row is an empty spacer row. Finally the third row holds the search, locale changer and optionally the administration link.

The first row starts off:

```
<!-- Welcome Message, Login/Logout -->
<af:panelGroupLayout id="pt_pg13" layout="horizontal">
  <f:facet name="separator">
    <af:spacer width="10" height="10" id="pt_s11"/>
  </f:facet>
```

The first row is represented by a `panelGroupLayout` container. The elements of the container are arranged horizontally (**`layout="horizontal"`**) and are separated with a 10-by-10 pixel spacer. The separator facet of the `panelGrouplayout` defines the separator element that is placed between all of the components in the container.

```
<!-- Welcome Message -->
<af:outputText id="ot_username"
    rendered="#{securityContext.authenticated}"
    value="#{portalBundle.welcome} : #{securityContext.userName}"/>
```

This is how the greeting is displayed. Please note that the whole greeting message is rendered only for authenticated users (`rendered="#{securityContext.authenticated}"`) and the message itself is not hardcoded, but is retrieved from the current resource bundle (`#{portalBundle.welcome}`). The name of the authenticated user is obtained from the security context (`#{securityContext.userName}`).

The next two elements display the login and logout buttons using ADF `commandButton`. Only one of the buttons is displayed, since they have opposite conditional values in the rendered attribute based on `securityContext.authenticated`.

```
<!-- Login/Logout -->
<af:commandButton id="cb_login"
```

```
        rendered="#{attrs.showLogin and
                    !securityContext.authenticated}"
        text="#{portalBundle.login}">
      <af:showPopupBehavior popupId="pt_p1" triggerType="click"/>
    </af:commandButton>
```

The login button activates a popup (`showPopupBehavior`) when clicked (`triggerType="click"`). Soon you will see how the popup implements the login functionality.

```
    <af:commandButton id="cb_logout"
        rendered="#{securityContext.authenticated}"
        action="#{o_w_s_l_LoginBackingBean.doLogout}"
        text="#{portalBundle.logout}"/>
```

Every WebCenter Portal Framework application contains a predefined managed bean, called: `o_w_s_l_LoginBackingBean`. This bean contains properties to pass username and password values and two methods to execute login and logout. The logout button calls the `doLogout` method of the backing bean (`action="#{o_w_s_l_LoginBackingBean.doLogout}"`).

```
  </af:panelGroupLayout>
```

Here is the second row, which is a simple 40 pixel high spacer.

```
  <af:spacer width="10" height="40" id="pt_s1"/>
```

Here is the third row:

```
  <!-- Global links: Search, Local changer, Admin Link -->
  <af:panelGroupLayout id="pt_pg14"
                       inlineStyle="white-space:nowrap"
                       layout="horizontal">
    <f:facet name="separator">
       <af:outputText value="|" id="pt_bar1"
                      styleClass="globalLinkSeparator" />
    </f:facet>
```

Similar to the first row, elements of this row are also arranged horizontally, separated by a spacer. Here the spacer is a vertical bar character, surrounded on each side by a 5 pixel space, added by the CSS style.

```
.globalLinkSeparator {
    margin: 0 5px;
}
```

```
    <!-- search -->
    <af:panelGroupLayout id="pt_pg16" layout="horizontal"
                         valign="top">
      <af:inputText id="it1" styleClass="searchtext" columns="10"/>
      <af:commandButton id="pt_cl4" text="#{portalBundle.search_text}"/>
    </af:panelGroupLayout>
```

The search component in the header is made up by an `inputText` and a `commandButton`. The `inputText` component's style (`searchtext`) gives it a rounded border. Earlier you could also see that the command button is styled.

```
<!-- Locale changer -->
<af:panelGroupLayout layout="horizontal" id="pt_pg17">
  <af:image source="#{images}/localizationMenuGlyph.png"
            id="pt_i1"/>
  <af:spacer width="3" height="10" id="pt_sp10"/>
  <af:selectOneChoice
        id="changelangsoc" autoSubmit="true"
        label="#{portalBundle.CHANGE_LOCALE}"
        valueChangeListener="#{ChangeLocale.listChanged}"
        styleClass="langsetter">
    <f:selectItems value="#{ChangeLocale.supportLocales}"
                   id="si_locals"/>
  </af:selectOneChoice>
</af:panelGroupLayout>
```

The user interface to change the locale contains an icon image (`localizationMenuGlyph.png`) and a drop-down list (`selectOneChoice`). The items displayed in this drop-down list are populated by a managed bean, `ChangeLocale`. Later you will see how this bean is implemented.

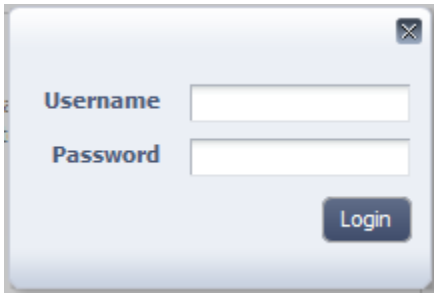
Each WebCenter Portal Framework application has a built-in administration page which is accessible by administrators only. The administration page has a fixed URL (`/<context root>/admin`).

```
<!-- Admin link -->
<af:goLink id="pt_glnk1" destination="/admin"
           text="#{portalBundle.admin}" styleClass="logintext"
           rendered="#{attrs.showAdmin}"/>
```

This `goLink` button provides access to the above URL. The button is rendered only when the `showAdmin` page attribute is true.

```
</af:panelGroupLayout>
```

Finally here is the pop-up component that is displayed when the user clicks on the login link:



The pop-up component's id, `pt_p1` was used earlier in the login button's `showPopupBehavior` tag.

```
<!-- logIn popup -->
<af:popup id="pt_p1">
```

```
<af:panelWindow id="pt_pw1" styleClass="loginPopupStyle"
                modal="true">
  <!-- Login form -->
  <af:subform id="pt_sf1" defaultCommand="pt_logincb">
    <af:panelFormLayout id="pt_pfl1">
```

The pop-up contains a `panelWindow`. Inside the window we use a special layout component, `panelFormLayout` that arranges its content in a two column grid.

```
<af:panelLabelAndMessage id="pt_plam1"
                         label="#{portalBundle.username}"
                         for="pt_it1"
                         styleClass="logintext">
  <af:inputText id="pt_it1"
                value="#{o_w_s_l_LoginBackingBean.userName}"
                columns="15"/>
</af:panelLabelAndMessage>
<af:panelLabelAndMessage id="pt_plam2"
                         label="#{portalBundle.password}"
                         for="pt_it2"
                         styleClass="logintext">
  <af:inputText id="pt_it2"
                value="#{o_w_s_l_LoginBackingBean.password}"
                columns="15" secret="true"/>
</af:panelLabelAndMessage>
```

The first two rows are made of a label and an input component. Note that the second row's input component, which is used for the password, has the attribute `secret="true"`. As you learned earlier, there is a predefined managed bean for implementing the login/logout functionality. The two `inputText` fields are associated with the username and password properties of this bean. For example: `value="#{o_w_s_l_LoginBackingBean.userName}"`.

```
<af:spacer width="10" height="5" id="pt_s14"/>
<af:panelGroupLayout id="pt_pg18" layout="horizontal"
                     halign="end">
  <af:commandButton
        id="pt_logincb"
        text="#{portalBundle.login}"
        action="#{o_w_s_l_LoginBackingBean.doLogin}"
        styleClass="logintext"/>
```

The third row's button is displayed right-aligned (`halign="end"`) and invokes the `doLogin` method of the managed bean.

```
    </af:panelGroupLayout>
  </af:panelFormLayout>
 </af:subform>
</af:panelWindow>
</af:popup>
```

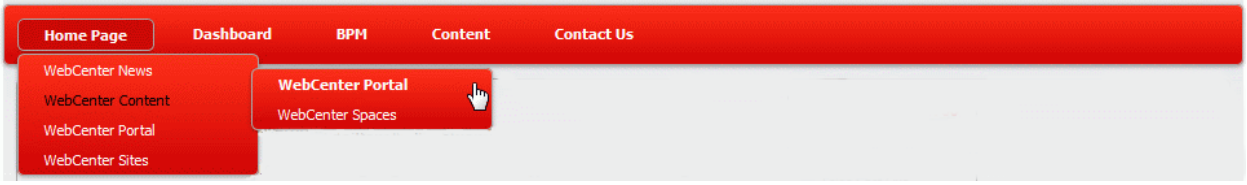
What remains from this part of the code are the closing elements:

```
</af:panelGroupLayout>
```

```
    </f:facet>
  </af:panelBorderLayout>
```

Rendering the Navigation Model

This is how the navigation bar looks:



Although WebCenter Portal Framework offers three task flows to render a navigation model as a menu, tree, or breadcrumb, here we decided to render it in a custom and dynamic way.

The navigation is rendered as a series of fly-out menus. The top level links are displayed horizontally and when the mouse hovers over a link, it will be rendered with a slightly darker bordered background. If there are sub-nodes in the navigation model, hovering over the parent node pops up a second or third vertical list of the page links on that level. Note that the currently rendered page's name is displayed with black characters and the link under the mouse is displayed in bold white characters.

```
<af:panelGroupLayout id="pt_navbar" layout="horizontal"
                     styleClass="navbarStyle AFStretchWidth"
                     rendered="#{showTemplate}">
```

As we have seen earlier, the navigation bar is rendered in a horizontal `panelGroupLayout` container. It has its own style and it is rendered only when the template has to be displayed.

It would be very difficult, if not impossible, to render such a fine-tuned dynamic navigation display using ADF components. Here we use a technique borrowed from contemporary Web page design - we use simple HTML tags enhanced with CSS elements. The pop-ups are implemented manipulating the `display` CSS property, displaying or hiding menus depending on the mouse position.

Some of dynamic page rendering also uses JavaScript libraries, like jQuery. Our example relies only on CSS definitions, but similar techniques could be used to incorporate JavaScript code.

In the page template we create a plain HTML fragment, where the lists of page links on every level are represented by unordered lists (`<ul>` and `<li>` tags), and the links themselves are anchors (`<a>` tags). Here is an example:

```
<div id="navbox">
  <ul id="nav" class="navbar">
    <li>
      <a ... />
      <ul>
        <li>
          <a  .../>
          <ul>
            <li><a  .../></li>
```

```
            <li><a  .../></li>
          </ul>
        </li>
        <li><a  .../></li>
      </ul>
    </li>
    <li><a ... /><li>
  <ul>
</div>
```

We use the outermost tag: `<div id="navbox">`, to group the other tags. Every level in the navigation model tree is represented as an unordered list; each element of this list is either an anchor or an embedded unordered list for the next level in the tree.

Here is the code that creates the HTML tags. Note that you can embed plain HTML tags inside an ADF page or page template, however sometimes ADF reorders the resulting HTML code. As a rule of thumb, group all of your plain HTML tags in a `<div>` and do not use any ADF components inside this block.

```
<div id="navbox">
  <c:set var="navNodes"
         value="${navigationContext.defaultNavigationModel.
         listModel['startNode=/, includeStartNode=false']}"
         scope="page"/>
```

First we add the outermost `<div>` tag, and then we create a temporary variable. The expression:

```
#{navigationContext.defaultNavigationModel.
   listModel['startNode=/, includeStartNode=false']}
```

returns from the default navigation model the list of nodes on the first level, i.e. starting at, but not including, the root node.

```
<ul id="nav" class="navbar">
```

The list of nodes on the root level is represented by an unordered list. The `navbar` style name is used when defining the other CSS styles.

```
<c:forEach var="menu" items="${navNodes}">
```

As we iterate through these nodes, the loop's current node will be placed in the `menu` variable.

```
<li>
  <a href="${contextRoot}${menu.goLinkPrettyUrl}"
     class="${menu.selected ? 'currentNode' : ''}">
    ${menu.title}</a>
```

We know that the current node is a page link so we create an anchor tag embedded in a list item. The `goLinkPrettyUrl` attribute of the current node returns the path for the page, which is suitable for the `href` attribute of the anchor. The expression `menu.selected` indicates if the page we are rendering is

the same as the current node in the navigation model. We use different styles to display the current and other pages in the navigation bar. The tag generated will be one of the following:

```
<a href href="url" class="currentNode">title</a>
<a href href="url" class="">title</a>
```

```
<c:if test="${not empty menu.children}">
```

We test if the current node has children in the navigation model. If there are children, we do the same as on the root level: iterate through the children, for every child node we open a list item tag (`<li>`), and create the anchor for the page link.

```
<ul>
  <c:forEach var="child" items="#{menu.children}">
    <li>
      <a href="${contextRoot}${child.goLinkPrettyUrl}"
         class="${child.selected ? 'currentNode' :
                                   ''}">
         ${child.title}</a>
```

The third level of the navigation, if it exists, is rendered similarly.

```
      <c:if test="${not empty child.children}">
        <ul>
          <c:forEach var="grandchild"
                     items="#{child.children}">
            <li>
              <a href=
            "${contextRoot}${grandchild.goLinkPrettyUrl}"
                 class="${grandchild.selected
                          ? 'currentNode' : ''}">
                 ${grandchild.title}</a>
            </li>
          </c:forEach>
        </ul>
      </c:if>
    </li>
  </c:forEach>
</ul>
</c:if>
</li>
```

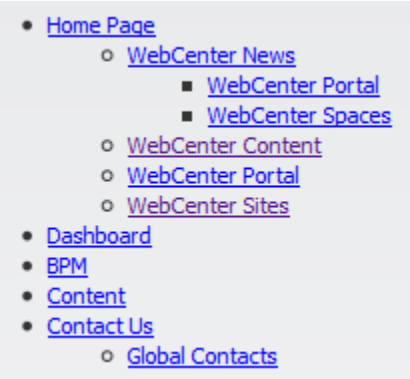
Here are the closing tags for the first level: the end of the loop, the end of the root unordered list, and the end of the `<div>` that contains all navigation tags.

```
    </c:forEach>
  </ul>
</div>
```

Here is the closing tag of the navigation bar.

```
</af:panelGroupLayout>
```

If we don't use additional CSS styles, our navigation model would be rendered like this:



Applying CSS styles allows completely different rendering results. Explaining the intricacies of the CSS styles is beyond the scope of this document, but we do provide a brief overview of the technique used here. For additional details, please consult the application skin file (`app-skin.css`), where all the CSS definitions reside. Here we list the important styles and give an explanation of their attribute settings.

The horizontal box surrounding the root nodes is rendered with the following style. Note that the style is applied to a node with the given identifier, in our case the first unordered list: `<ul id="nav" class="navbar">`.

```
.navbar { ... a rounded box with gradient redish background and a given height ... }
```

All the other style definitions apply to HTML tags below the `nav` node. The list items on the first level, i.e. in the navigation bar, are rendered as follows:

```
.navbar li { ... the items are rendered horizontally, without bullets ... }
```

We render the list items in the sub-menus differently. We change the direction that the list items are rendered:

```
.navbar ul li { ... the items rendered vertically ... }
```

The anchors in the first level are rendered as follows. The second definition applies to the current page, and changes the color of the link to black.

```
.navbar a { ... display the text in white, bold characters ... }
.navbar a.currentNode { color: black !important; }
```

The anchors in the first level are displayed differently, when the mouse hovers over the list item:

```
.navbar li:hover a { ... add a darker rounded box around the anchor text ... }
```

The displayed anchor changes when the mouse hovers over a list item in a sub-menu. We remove the background and the bordered box added in the root level.

```
.navbar ul li:hover a, #nav li:hover li a {
   ... remove border and background, set the font style to normal ...
}
```

The child list (an `<ul>` tag below the `nav` node) is displayed as follows:

```
.navbar ul { ... a hidden rounded box with gradient background,
             positioned bellow the current coordinates ... }
```

The style for the second level child list inherits these settings, but changes the display position:

```
.navbar ul ul { ... change the position to the right of the current element ... }
```

When the mouse hovers over a list item, if there is a list (`<ul>`) for sub-menu, we change its `display` property.

```
.navbar li:hover > ul { display: block; }
```

Page Content

```
<af:panelBorderLayout id="pt_content" styleClass="contentStyle">
  <!-- content facet -->
  <af:facetRef facetName="content"/>
</af:panelBorderLayout>
```

When rendering an application page, the actual page content will replace the `facetRef` tag. We placed this tag into a container to provide a style (`contentStyle`).

```
af|panelBorderLayout.contentStyle {
    margin: 5px 10px;
    border-radius: 5px;
    border: solid #ABABAB 1.5px;
}
```

This style renders a rounded thin border and adds some margin around the page content.

Page Footer

This is how the page footer looks:


```
<af:panelGroupLayout id="pt_pg_footer" layout="vertical"
                     halign="center" styleClass="footerStyle"
                     rendered="#{showTemplate}">
```

As we have seen earlier, the page footer is rendered by a vertical `panelGroupLayout` container. It has its own style and it is rendered only when the template has to be displayed. The footer is made of three rows: a list of external links, a horizontal separator bar and the copyright text.

```
<af:panelGroupLayout id="pt_pg23" layout="horizontal"
                     halign="center">
```

The first row is a horizontal, centered `panelGroupLayout` component. Inside this container we render the global navigation model, similar to the way the default navigation was rendered.

```
  <af:forEach
     var="node" varStatus="vs"
     items="#{navigationContext.navigationModel
['modelPath=/oracle/webcenter/portalapp/navigations/global-navigation-
model'].listModel['startNode=/, includeStartNode=false']}">
```

The loop is similar to the loop in the navigation bar, but instead of the default navigation model, here we use a specific navigation model at `/oracle/webcenter/portalapp/navigations/global-navigation-model`.

```
    <af:goLink id="cl2" styleClass="footerText"
               text="#{node.title}"
               destination="#{node.goLinkPrettyUrl}"/>
    <af:outputText value="|"
                   id="pt_bar2" styleClass="footerLinkSeparator"
                   rendered="#{!vs.last}"/>
```

In the loop we use a `goLink` and a vertical separator. The separator is not rendered after the last link.

```
  </af:forEach>
</af:panelGroupLayout>



<af:panelGroupLayout
      id="pt_pg25" layout="horizontal"
      styleClass="AFStretchWidth"
      inlineStyle="background-color:white; margin:5px 0;">
  <af:spacer width="10" height="1" id="pt_s9"/>
```

The second row is also a horizontally centered `panelGroupLayout` component. The container contains a spacer element, which is a small, white horizontal line. This line is stretched to the width of the page (`styleClass="AFStretchWidth"`), excluding previously defined margins.

```
</af:panelGroupLayout>



<af:panelGroupLayout id="pt_pg26" layout="horizontal" halign="center">
  <af:outputText id="pt_copyright" value="#{portalBundle.copyright}"
                 inlineStyle="color:white;font-size:x-small;"/>
</af:panelGroupLayout>
```

The copyright message is displayed in its own horizontal, centered `panelGroupLayout` container. The message is coming from the resource bundle.

```
</af:panelGroupLayout>
```

A final note about the page template: in order for a page template to be editable at run time, the template must include editable components - `panelCustomizable` and `showDetailFrame`. In our example we do not use these elements so this page template will not be editable at run time.

Using the Page Template

In this section, we briefly discuss how the previously created template can be used in the WebCenter Portal Framework application.

Create Portal Resource from the Template

We recommend that you create a portal resource from your template. It is not strictly necessary, but creating a portal resource gives you some advantages.

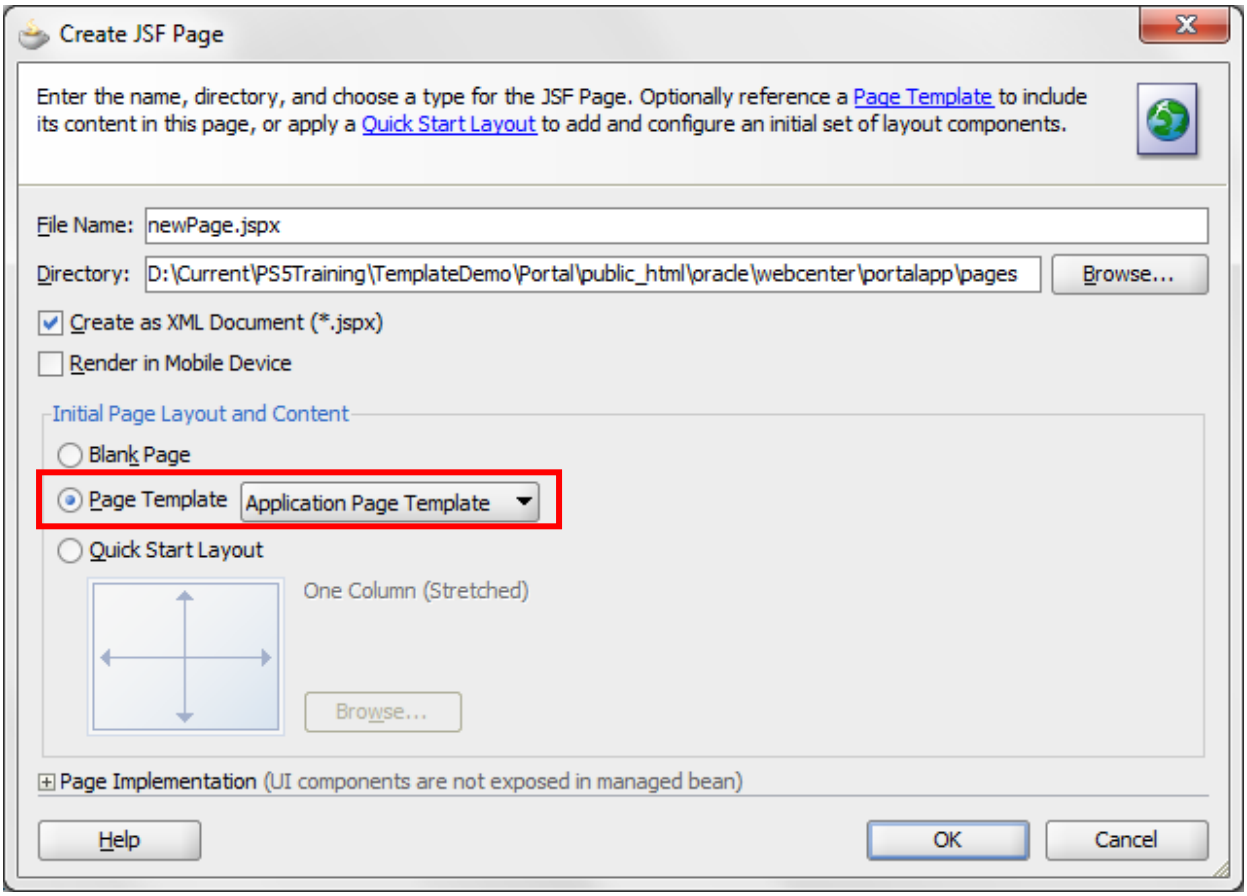
Portal resources:

- can be packaged into resource archives. These archives can be imported and used in other WebCenter Portal Framework applications. You can create general purpose, reusable page templates.
- can be managed at run time using the Administration Console.

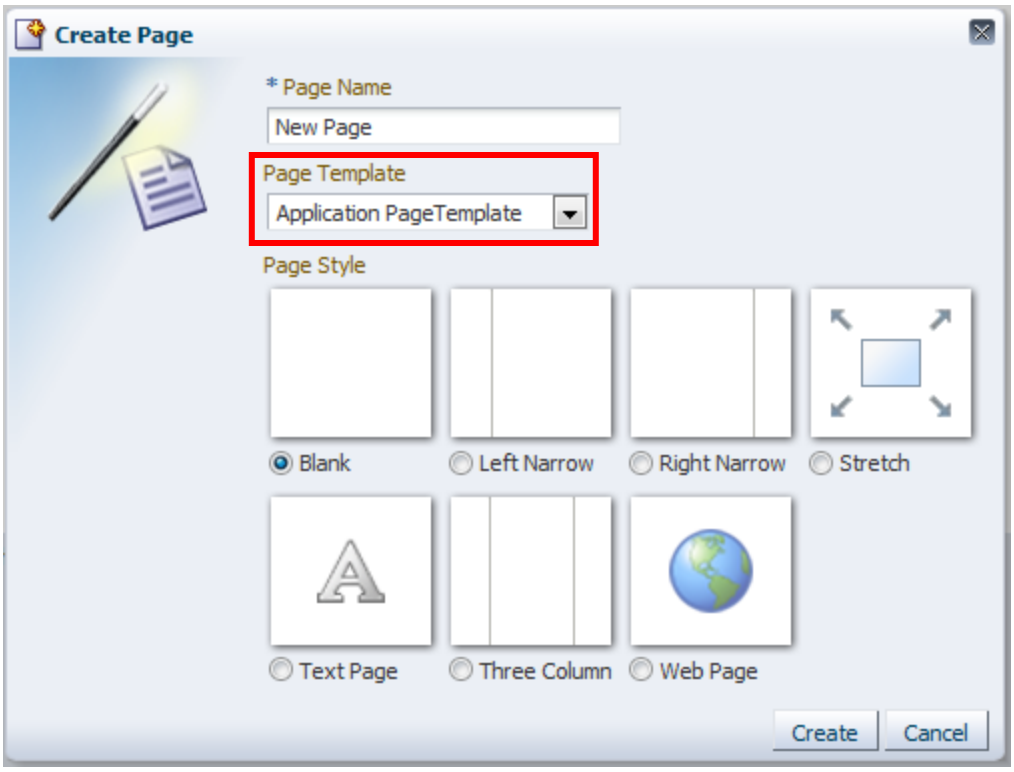
For details on creating portal resources, see the slides of the related training module. In the rest of this document we will not create portal resources.

Create Pages Using the Template

Once you have created (or imported) a page template you can use it when creating new pages. At design time (JDeveloper) the wizard lets you select the page template.



You can also choose the page template at run time when you create a new page by using the Administration Console.



Here is the page source that was created in JDeveloper, using our example template: *Application Page Template*.

```
<?xml version='1.0' encoding='UTF-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
  <jsp:directive.page contentType="text/html;charset=UTF-8"/>
  <f:view>
    <af:document id="d1">
      <af:form id="f1">
        <af:pageTemplate
viewId="/oracle/webcenter/portalapp/pagetemplates/AppPageTemplate.jspx"
            value="#{bindings.pageTemplateBinding}" id="pt1">
          <f:facet name="content"/>
        </af:pageTemplate>
      </af:form>
    </af:document>
  </f:view>
</jsp:root>
```

And here is the corresponding page definition file, which contains the component bindings of the page. The page template related binding is highlighted:

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="11.1.1.62.29" id="newPagePageDef"
                Package="oracle.webcenter.portalapp.pages">
```

```
    <parameters/>
    <executables>
      <variableIterator id="variables"/>
      <page
       path="oracle.webcenter.portalapp.pagetemplates.AppPageTemplatePageDef"
       id="pageTemplateBinding" Refresh="ifNeeded"/>
    </executables>
    <bindings/>
  </pageDefinition>
```

As you can see, the code is hardly more than a reference to the page template. Inside the `pageTemplate` tag is an empty content facet. Your task will be to add the page content into the content facet.

There is one important feature of this code; it uses a hardcoded page template by referencing the page template file `AppPageTemplate.jspx`. In many cases, this is acceptable, but it misses one of the features of WebCenter Portal applications - the application configuration, such as default page template, default skin, etc., can be changed dynamically. Note that the default values can be changed at design time by modifying `adf-config.xml` file (see above when the skin file was discussed) and at run time by using the Administration Console's Configuration tab.

JDeveloper's Create New Page wizard creates this code, so in order to utilize the dynamic configuration feature; you have to manually change the source code. In the page source file, replace the `pageTemplate` tag with the following code:

```
<af:pageTemplate value="#{bindings.pageTemplateBinding.templateModel}"
                 id="pt1">
  <f:facet name="content"/>
</af:pageTemplate>
```

and in the page definition file, modify the page tag as follows:

```
<page viewId="${preferenceBean.defaultPageTemplate}"
      id="pageTemplateBinding" Refresh="ifNeeded"/>
```

As you can see in the modified source, `viewId` is not hardcoded in the page source, but it is obtained from the `defaultPageTemplate` property of the predefined `preferenceBean` managed bean.

What Should Go into a Page

It is up to the application developer what content he or she adds to the pages. However we would like to point out two things to consider:

1. Keep in mind that the page template we developed uses a flow-type layout. It means that whatever you place in your page is surrounded by layout containers that set the width of the page and do not let you use stretching containers, like `panelStretchLayout` or `panelSplitter`. In the related training module, you can get good suggestions regarding what to use and what not to use inside a flowing layout.
2. If you want to make your page editable at runtime, use editable components. Surround your page with a `pageCustomizable` component and add `panelCustomizable` components, wherever you want

the page to be editable. Here is an example of an editable but otherwise empty page. Note the highlighted elements.

```
<?xml version='1.0' encoding='UTF-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
          xmlns:pe="http://xmlns.oracle.com/adf/pageeditor"
          xmlns:cust="http://xmlns.oracle.com/adf/faces/customizable"
          xmlns:c="http://java.sun.com/jsp/jstl/core">
  <jsp:directive.page contentType="text/html;charset=UTF-8"/>
  <f:view>
    <af:document id="d1">
      <af:form id="f1">
        <af:pageTemplate
              value="#{bindings.pageTemplateBinding.templateModel}"
              id="pt1">
          <f:facet name="content">
            <pe:pageCustomizable id="hm_pgc1">
              <cust:panelCustomizable id="hm_pnc1" layout="auto">

                … Here comes the page content …

              </cust:panelCustomizable>
              <f:facet name="editor">
                <pe:pageEditorPanel id="pep1"/>
              </f:facet>
            </pe:pageCustomizable>
          </f:facet>
        </af:pageTemplate>
      </af:form>
    </af:document>
  </f:view>
</jsp:root>
```

About Page Styles

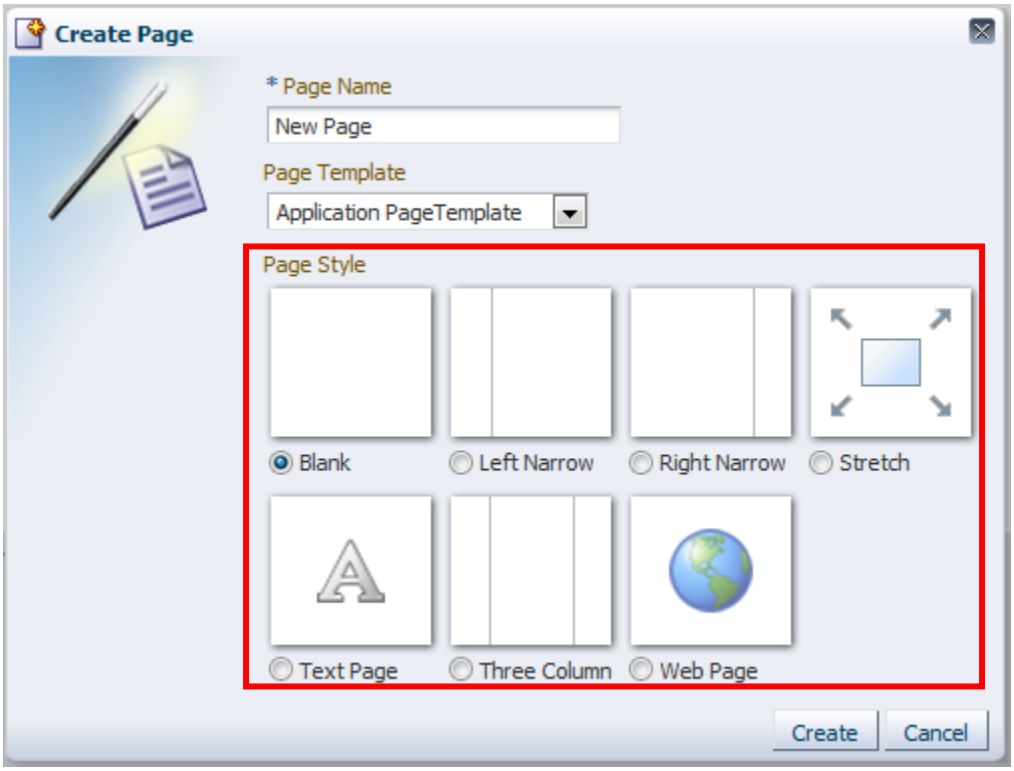
Although a page template provides a general structure of the application's pages, it is typical that several of the pages' content also share a common structure. For example:

- We want to create editable pages. As you saw above, it means that each of these pages should include editable components. Note that these components, especially `pageCustomizable`, cannot be placed in the page template; it must be included in the page, i.e., inside the content facet.
- The content of the pages would require a similar layout. For example, we would like to arrange the content in 2 or 3 columns. Although technically it would be possible to create a page template with several facets, each within its own column, but as we discussed above, the run-time tools of a Portal Framework application recognize only a single facet called content.

All these use cases would require creating pages with some initial content.

However if you create your pages at run time, WebCenter Portal applications can help. There is a resource type called page style, which provides initial content for the pages.

When you create a page at run time, you can choose one page style to populate the initial page:



For example, the *Blank* style contains the necessary editable components to make your pages editable. The *Left Narrow*, *Right Narrow* and *Three Column* page styles in addition to the editable components contain layout components to arrange your content into columns.

You can also develop and register your own page style. The steps are simple:

1. Create your example page that contains the common elements you want to include in the style. You should create it as you would create any application page: use a page template, and then change the code to refer to the default page template. Add your components.
2. Create a portal resource from this page. JDeveloper recognizes that this JSPX file is not a page template, so it creates a page style resource.
3. If needed, you can package the resource and upload it into other applications.

Keep in mind that page styles are usable only at run time. When creating pages in JDeveloper, copy/paste is currently the only option.

Internationalizing the Page Template

Internationalization is the process of designing an application so that it can be adapted to various languages and regions without engineering changes. Internationalizing WebCenter Portal Framework applications is a complex topic on its own. In this document we merely aim is to discuss how this page template is created with internationalization techniques in mind. It builds on standard Java internationalization techniques.

Important concepts:

- All the static texts in the template are stored in resource bundle files. If a new language needs to be supported, create a new resource bundle with the translated texts.
- The run-time environment maintains the current locale. The locale defines the localized environment: the language, and the language variant to be used. Note that for the sake of simplicity we do not distinguish language variants.
- We provide a mechanism to set and change the application's preferred locale.
- Java Virtual machine picks up the correct resource bundle, depending on the current locale. If the current locale is not supported, i.e. there is no translated resource bundle for the locale's language; the default resource bundle is selected.
- All the static texts in the page template are fetched and displayed from the current resource bundle. See the discussion of the resource bundle files above.

Let's see some of the details. We created two supporting Java classes. You can see the complete source code when you download the example.

- `templatedemo.portal.ChangeLocale`

  This Java class provides support for changing the locale. The class is registered as a managed bean (`ChangeLocale`) in the session scope. In `faces-config.xml` here is the bean registration tag:

```
<managed-bean>
  <managed-bean-name>ChangeLocale</managed-bean-name>
  <managed-bean-class>templatedemo.portal.ChangeLocale</managed-bean-
class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

  The bean provides a property `preferredLocale` with getter and setter methods to access the selected locale.

  There is also a property (`supportLocales`) to return the supported locales in the environment as drop-down list items and an event handler method (`listChanged`) which is called, when a locale is selected from the drop-down list. Here is how these methods are used in the page header banner.

```
<af:selectOneChoice id="changelangsoc" autoSubmit="true"
                    label="#{portalBundle.CHANGE_LOCALE}"
                    valueChangeListener="#{ChangeLocale.listChanged}"
                    styleClass="langsetter">
```

```
  <f:selectItems value="#{ChangeLocale.supportLocales}"
                 id="si_locals"/>
</af:selectOneChoice>
```

The event handler not only saves the selected locale, but also refreshes the current page by redirecting the browser to the same URL. Of course, when the page is rendered again, the newly selected locale is used.

A note on `supportLocales`: currently the list of supported locales is provided by the environment. Inside the method you can find this code fragment:

```
FacesContext.getCurrentInstance().getApplication().getSupportedLocales(
)
```

It means that some of locales listed have no corresponding translated resource bundle file. It is not a big problem since selecting a locale without a translation uses the default resource bundle, but the list shows misleading information to the user.

- `templatedemo.portal.CustomPhaseListener`

  This is an ADF phase listener. It is registered in the `adf-settings.xml` file:

```
<lifecycle>
  <phase-listener>
      <listener-id>MyAdfListener</listener-id>
      <class>templatedemo.portal.CustomPhaseListener</class>
  </phase-listener>
</lifecycle>
```

  Our custom listener is active before ADF's prepare model phase, before the page's internal representation is created. The listener has two functions: it ensures that the current locale is set for the user interface, and it invalidates the model cache. That ensures that the page is created in the correct locale.

We have to define a variable through which the current resource bundle file can be accessed. You can do it in two places:

- Set the default bundle globally in `faces-config.xml`. We do not use this possibility in our application.

```
<resource-bundle>
  <base-name>pocdemo.portal.PortalBundle</base-name>
  <var>portalBundle</var>
</resource-bundle>
```

- Set the bundle in all of the pages, or page templates:

```
<c:set var="portalBundle"
       value="#{adfBundle['templatedemo.portal.PortalBundle']}"/>
```

This approach has the advantage that we can provide individual translation to every page if we define different resource bundle files. If we have a single resource file for the whole application, this file gets very large, containing all the texts of all the application pages.

Appendix A: Source for the Page Template: `AppPageTemplate.jspx`

```
<?xml version='1.0' encoding='UTF-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
          xmlns:pe="http://xmlns.oracle.com/adf/pageeditor"
          xmlns:cust="http://xmlns.oracle.com/adf/faces/customizable"
          xmlns:c="http://java.sun.com/jsp/jstl/core">
  <c:set var="portalBundle"
         value="#{adfBundle['templatedemo.portal.PortalBundle']}"/>
  <c:set var="contextRoot"
         value="${facesContext.externalContext.requestContextPath}"
         scope="session"/>
  <c:set var="images"
         value="/${contextRoot}/oracle/webcenter/portalapp/shared/pagetemplates/images"
         scope="session"/>
  <c:set var="showTemplate"
         value="${!composerContext.inEditMode or attrs.isEditingTemplate}"
         scope="session"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8"/>
  <af:pageTemplateDef var="attrs">
    <!-- Main container -->
    <af:panelGroupLayout id="pt_root" layout="scroll"
                         inlineStyle="margin: 5px 0;">
      <!-- centered vertical stack: banner, navbar, content, footer -->
      <af:panelGroupLayout id="pt_main" layout="vertical" styleClass="mainStyle"
                           inlineStyle="width:#{attrs.contentWidth}; margin: 0 auto;">
        <!-- ***** banner ***** -->
        <af:panelBorderLayout id="pt_banner" styleClass="bannerStyle AFStretchWidth"
                              rendered="#{showTemplate}">
          <!-- Logo image -->
          <f:facet name="start">
              <af:goLink id="home" destination="/">
                <af:image source="#{images}/logo.png"/>
              </af:goLink>
          </f:facet>
          <f:facet name="end">
            <af:panelGroupLayout layout="vertical" id="pt_pg12" halign="end"
                                 inlineStyle="padding: 15px 0px">
              <!-- Welcome Message, Login/Logout -->
              <af:panelGroupLayout id="pt_pg13" layout="horizontal">
                <f:facet name="separator">
                  <af:spacer width="10" height="10" id="pt_s11"/>
                </f:facet>
                <!-- Welcome Message -->
                <af:outputText id="ot_username"
                               rendered="#{securityContext.authenticated}"
                               value="#{portalBundle.welcome} : #{securityContext.userName}"/>
                <!-- Login/Logout -->
                <af:commandButton id="cb_login"
                                  rendered="#{attrs.showLogin and
                                              !securityContext.authenticated}"
                                  text="#{portalBundle.login}">
                  <af:showPopupBehavior popupId="pt_p1" triggerType="click"/>
                </af:commandButton>
                <af:commandButton id="cb_logout"
                                  rendered="#{securityContext.authenticated}"
                                  action="#{o_w_s_l_LoginBackingBean.doLogout}"
                                  text="#{portalBundle.logout}"/>
              </af:panelGroupLayout>
              <af:spacer width="10" height="40" id="pt_s1"/>
              <!-- Global links: Search, Local changer, Admin Link -->
              <af:panelGroupLayout id="pt_pg14"
                                   inlineStyle="white-space:nowrap"
                                   layout="horizontal">
                <f:facet name="separator">
                   <af:outputText value="|" id="pt_bar1" styleClass="globalLinkSeparator" />
                </f:facet>
                <!-- search -->
```

```
        <af:panelGroupLayout id="pt_pg16" layout="horizontal"
                             valign="top">
          <af:inputText id="it1" styleClass="searchtext" columns="10"/>
          <af:commandButton id="pt_cl4" text="#{portalBundle.search_text}"/>
        </af:panelGroupLayout>
        <!-- Locale changer -->
        <af:panelGroupLayout layout="horizontal" id="pt_pg17">
          <af:image source="#{images}/localizationMenuGlyph.png"
                    id="pt_i1"/>
          <af:spacer width="3" height="10" id="pt_sp10"/>
          <af:selectOneChoice id="changelangsoc" autoSubmit="true"
                              label="#{portalBundle.CHANGE_LOCALE}"
                              valueChangeListener="#{ChangeLocale.listChanged}"
                              styleClass="langsetter">
            <f:selectItems value="#{ChangeLocale.supportLocales}"
                           id="si_locals"/>
          </af:selectOneChoice>
        </af:panelGroupLayout>
        <!-- Admin link -->
        <af:goLink id="pt_glnk1" destination="/admin"
                   text="#{portalBundle.admin}" styleClass="logintext"
                   rendered="#{attrs.showAdmin}"/>
      </af:panelGroupLayout>
      <!-- logIn popup -->
      <af:popup id="pt_p1">
        <af:panelWindow id="pt_pw1" styleClass="loginPopupStyle" modal="true">
          <!-- Login form -->
          <af:subform id="pt_sf1" defaultCommand="pt_logincb">
            <af:panelFormLayout id="pt_pfl1">
              <af:panelLabelAndMessage id="pt_plam1"
                                       label="#{portalBundle.username}"
                                       for="pt_it1"
                                       styleClass="logintext">
                <af:inputText id="pt_it1"
                              value="#{o_w_s_l_LoginBackingBean.userName}"
                              columns="15"/>
              </af:panelLabelAndMessage>
              <af:panelLabelAndMessage id="pt_plam2"
                                       label="#{portalBundle.password}"
                                       for="pt_it2"
                                       styleClass="logintext">
                <af:inputText id="pt_it2"
                              value="#{o_w_s_l_LoginBackingBean.password}"
                              columns="15" secret="true"/>
              </af:panelLabelAndMessage>
              <af:spacer width="10" height="5" id="pt_s14"/>
              <af:panelGroupLayout id="pt_pg18" layout="horizontal"
                                   halign="end">
                <af:commandButton id="pt_logincb"
                                  text="#{portalBundle.login}"
                                  action="#{o_w_s_l_LoginBackingBean.doLogin}"
                                  styleClass="logintext"/>
              </af:panelGroupLayout>
            </af:panelFormLayout>
          </af:subform>
        </af:panelWindow>
      </af:popup>
    </af:panelGroupLayout>
  </f:facet>
</af:panelBorderLayout>
<!-- ***** navbar ***** -->
<af:panelGroupLayout id="pt_navbar" layout="horizontal"
                     styleClass="navbarStyle AFStretchWidth"
                     rendered="#{showTemplate}">
  <!-- iterate over navigation items -->
  <div id="navbox">
    <c:set var="navNodes"
           value="${navigationContext.defaultNavigationModel.listModel
                    ['startNode=/, includeStartNode=false']}"
           scope="page"/>
    <ul id="nav" class="navbar">
```

```
            <c:forEach var="menu" items="${navNodes}">
              <li>
                <a href="${contextRoot}${menu.goLinkPrettyUrl}"
                   class="${menu.selected ? 'currentNode' : ''}">
                  ${menu.title}</a>
                <c:if test="${not empty menu.children}">
                  <ul>
                    <c:forEach var="child" items="#{menu.children}">
                      <li>
                        <a href="${contextRoot}${child.goLinkPrettyUrl}"
                           class="${child.selected ? 'currentNode' : ''}">
                           ${child.title}</a>
                        <c:if test="${not empty child.children}">
                          <ul>
                            <c:forEach var="grandchild"
                                       items="#{child.children}">
                              <li>
                                <a href="${contextRoot}${grandchild.goLinkPrettyUrl}"
                                   class="${grandchild.selected ? 'currentNode' : ''}">
                                   ${grandchild.title}</a>
                              </li>
                            </c:forEach>
                          </ul>
                        </c:if>
                      </li>
                    </c:forEach>
                  </ul>
                </c:if>
              </li>
            </c:forEach>
          </ul>
        </div>
      </af:panelGroupLayout>
      <!-- ***** content ***** -->
      <af:panelBorderLayout id="pt_content" styleClass="contentStyle">
        <!-- content facet -->
        <af:facetRef facetName="content"/>
      </af:panelBorderLayout>
      <!-- ***** footer ***** -->
      <af:panelGroupLayout id="pt_footer" layout="vertical" halign="center"
                           styleClass="footerStyle AFStretchWidth"
                           rendered="#{showTemplate}">
        <af:panelGroupLayout id="pt_pg23" layout="horizontal">
          <af:forEach var="node" varStatus="vs"
items="#{navigationContext.navigationModel['modelPath=/oracle/webcenter/portalapp/navigations/glo
bal-navigation-model'].listModel['startNode=/, includeStartNode=false']}">
            <af:goLink id="cl2" styleClass="footerText"
                       text="#{node.title}"
                       destination="#{node.goLinkPrettyUrl}"/>
            <af:outputText value="|" id="pt_bar2"
                           styleClass="footerLinkSeparator" rendered="#{!vs.last}"/>
          </af:forEach>
        </af:panelGroupLayout>
        <af:panelGroupLayout id="pt_pg25" layout="horizontal"
                             styleClass="footerSpacer">
          <af:spacer width="10" height="1" id="pt_s9"/>
        </af:panelGroupLayout>
        <af:panelGroupLayout id="pt_pg26" layout="horizontal">
          <af:outputText id="pt_copyright" styleClass="footerText"
                         value="#{portalBundle.copyright}"/>
        </af:panelGroupLayout>
      </af:panelGroupLayout>
    </af:panelGroupLayout>
  </af:panelGroupLayout>
  <af:xmlContent>
    <component xmlns="http://xmlns.oracle.com/adf/faces/rich/component">
      <display-name>Application Page Template</display-name>
      <facet>
        <description>Facet for content</description>
        <facet-name>content</facet-name>
      </facet>
```

```
        <attribute>
          <attribute-name>contentWidth</attribute-name>
          <attribute-class>java.lang.String</attribute-class>
          <default-value>960px</default-value>
        </attribute>
        <attribute>
          <attribute-name>showNavigation</attribute-name>
          <attribute-class>java.lang.Boolean</attribute-class>
          <default-value>#{true}</default-value>
        </attribute>
        <attribute>
          <attribute-name>showGreetings</attribute-name>
          <attribute-class>java.lang.Boolean</attribute-class>
          <default-value>#{securityContext.authenticated}</default-value>
        </attribute>
        <attribute>
          <attribute-name>showLogin</attribute-name>
          <attribute-class>java.lang.Boolean</attribute-class>
          <default-value>#{true}</default-value>
        </attribute>
        <attribute>
          <attribute-name>showAdmin</attribute-name>
          <attribute-class>java.lang.Boolean</attribute-class>
          <default-value>#{WCSecurityContext.userInAppRole['Administrator']}</default-value>
        </attribute>
        <attribute>
          <attribute-name>isEditingTemplate</attribute-name>
          <attribute-class>java.lang.Boolean</attribute-class>
          <default-value>#{false}</default-value>
        </attribute>
      </component>
    </af:xmlContent>
  </af:pageTemplateDef>
</jsp:root>
```

Appendix B: Source for the Application Skin: `app-skin.css`

```
/* ***** Template area styles ***** */

.mainStyle {
    background: #E6E7E9;
}

.bannerStyle {
    padding: 0 10px;
    border: 1px solid #E6E7E9;
    border-top: none;
    border-bottom: 2px solid #ABABAB;
    background-image:
      url('/oracle/webcenter/portalapp/shared/pagetemplates/images/topheaderbackground.png');
}

.navbarStyle {}

.contentStyle {
    margin: 5px 10px;
    border-radius: 5px;
    border: solid #ABABAB 1.5px;
}

.footerStyle {
    padding: 5px 10px;
    background: rgb(96, 108, 136);
}

.globalLinkSeparator {
    margin: 0 5px;
}

.footerLinkSeparator {
    margin: 0 10px;
    color: white;
    font-weight: bold;
}

.footerText {
    color: white;
    font-size: x-small;
    text-decoration:none;
}

.footerSpacer {
  width: 100%;
  background-color: white;
  margin:5px 0;
}

/* login popup */
af|panelWindow.loginPopupStyle::title {
    color: black;
    padding-left: 10px;
}

af|commandLink.logintext, af|panelLabelAndMessage.logintext::label {
    color: #4B5875;
    white-space: nowrap;
    font-weight: bold;
}

af|panelWindow.demopanelwindow::close-icon-style {
    background-image:
      url('/oracle/webcenter/portalapp/shared/pagetemplates/images/popClose_30x30.png');
    line-height: 10px;
    position: absolute;
    right: -10px;
```

```
    top: -10px;
    height: 30px;
    width: 30px;
    outline: 0;
}

/* global button settings */

.AFButtonBackground:alias {
    background-image:none;
    background-color: rgb(96, 108, 136);
}
.AFButtonBackgroundHover:alias {
    background-image: none;
    background-color: rgb(40, 52, 59);
}
.AFButtonBackgroundActive:alias {
    background-image:none;
    background-color: rgb(96, 108, 136);
}
.AFButtonBackgroundFocus:alias {
    background-image:none;
    background-color: rgb(96, 108, 136);
}
.AFButtonBorder:alias {
    border: solid #ABABAB 1px;
    border-radius: 5px;
}
.AFButtonBorderHover:alias {
    border:solid red 1px;
}
.AFButtonForeground:alias {
    text-align: center;
    white-space: nowrap;
    color: white;
}
.AFButtonForegroundHover:alias {
    color: white;
}

/* search box */
af|inputText.searchtext af|inputText::content {
    border: solid #ABABAB 1.5px;
    border-radius: 5px 0px 0px 5px;
    background-image: none;
    height: 17px;
    font-size: smaller;
}

/* locale setter */
af|selectOneChoice.langsetter::content {
    border: solid #ABABAB 1.5px;
    border-radius: 5px;
    background-image: none;
    height: 21px;
    white-space: nowrap;
}
af|selectOneChoice.langsetter::label {
    color: #4B5875;
    font-size: x-small;
    font-weight:bold;
}

/* ***** Misc. settings ***** */
/* Remove default gradient background from the page */
af|document {
  background-image: none;
  background-color: white;
}

/* Stretch the editable part of the page to the full width when in edit mode */
```

```
af|pageCustomizable:edit {
    width: auto;
    height: 1000px;
}

/* image styles for the pages content */
af|image.sidebarimg {
    width:160px;
}
af|image.mainwcimg {
    width:  400px;
    height: 325px;
}

/* ***** Flyout menu ***** */

/* Navgation bar */
.navbar {
    z-index: 130;
    height: 35px;
    margin: 5px 0px;
    padding: 5px;
    line-height: 100%;
    background: #ff3019;
    border-radius: 5px;
    border: solid #ABABAB 1px;
    box-shadow: 0 1px 3px rgba(0, 0, 0, .4);
}

.navbar li {
    padding: 5px;
    float: left;
    position: relative;
    list-style: none;
}

/* Links in the navigation bar */
.navbar a {
    font-weight: bold;
    color: white;
    text-decoration: none;
    display: block;
    padding:  6px 20px;
}

/* Link for the current node */
.navbar a.currentNode {
  color: black !important;
}

/* Hover over links in the navigation bar */
.navbar li:hover a {
    background: cc0000;
    border-radius: 5px;
    border: solid #ABABAB 1px;
    box-shadow: 0 1px 1px rgba(0, 0, 0, .2);
}

/* Hover over links in the child menus */
.navbar ul li:hover a, .navbar li:hover li a {
    background: none;
    border: none;
    box-shadow: none;
    font-weight:normal;
}

/* Child navigation list */
.navbar ul {
    display: none;

    position: absolute;
```

```
    width: 185px;
    top: 32px;
    left: 5px;
    margin: 0px;
    padding: 0px;
    background: #ff3019;
    border: solid #ABABAB 1px;
    border-radius: 5px;
    box-shadow: 0 1px 3px rgba(0, 0, 0, .3);
}

/* Popu-up the list, i.e. show list, when hoover over parent */
.navbar li:hover > ul {
    display: block;
}

/* Elements in the child lists */
.navbar ul li {
    float: none;
    margin: 0px;
    padding: 0px;
}

/* Hover over links in the child pop-ups */
.navbar ul a:hover {
    font-weight: bold !important;
}

/* Level 3+ lists positioning */
.navbar ul ul {
    top: 10px;
    left: 180px;
}

/* ***** End of flyout menu ***** */

/* ***** Browser-specific modifiers ***** */

@agent mozilla {
    .mainStyle {
        background: -moz-linear-gradient(top, #E6E7E9, #FFFFFF);
    }

    .footerStyle {
        background: -moz-linear-gradient(top, rgb(96, 108, 136) 0%, rgb(63, 76, 107)100%);
    }

    .navbar {
        background: -moz-linear-gradient(top, #ff3019 0%, #cf0404 100%);
    }

    .navbar li:hover a {
        background: -moz-linear-gradient(top, #cc0000 0%, #cc0000 100%);
    }

    .navbar ul {
        background: -moz-linear-gradient(top, #ff3019 0%, #cf0404 100%);
    }
}

@agent webkit {
    .mainStyle {
        background: -webkit-gradient(linear, center top, center bottom,
                      color-stop(0%, #E6E7E9), color-stop(100%, #FFFFFF));
    }

    .footerStyle {
        background: -webkit-gradient(linear, center top, center bottom,
                      color-stop(0%, rgb(96, 108, 136)), color-stop(100%, rgb(63, 76, 107)));
    }
```

```
    .navbar {
        background: -webkit-gradient(linear, center top, center bottom,
                    color-stop(0%, #ff3019), color-stop(100%, #cf0404));
    }

    .navbar li:hover a {
        background: -webkit-gradient(linear, center top, center bottom,
                    color-stop(0%, #cc0000), color-stop(100%, #cc0000));
    }

    .navbar ul {
        background: -webkit-gradient(linear, center top, center bottom,
                    color-stop(0%, #ff3019), color-stop(100%, #cf0404));
    }
}
```