MADE FROM THE SAME MOLD: TEMPLATING APPROACHES FOR ADF FACES APPLICATIONS

Peter Koletzke, Quovera

Users will more easily understand your application and will therefore be more productive if all pages of your web application are designed to look and act the same. If UI consistency is properly implemented, users can learn the common features of one page and they will quickly understand all other pages in the application. This consistency is so important that "common look and feel" is often part of the stated or implied requirements for new systems; this common look and feel has traditionally been met by the use of templates. An additional benefit of template use is easier maintenance; changing common elements in a number of windows or pages is often as easy as modifying a common file that is used by all windows or pages.

For example, when you set up a template system for Oracle Forms, you start by creating an object library filled with common objects and code. You then create a template FMB file that references the object library objects and includes attached code libraries for features common to your application such as help systems, messaging systems, calendar windows, shuttle controls, and so on. Each new form starts by copying this template FMB so that each new form automatically contains common look and feel elements and common functionality. Using templates for Forms applications means you have less work to do in the form; much functionality has been developed into the template so you just need to customize this functionality for the specific business requirements.

In addition, common look-and-feel aspects available in the object library, such as Smart Class objects and code, will be available to the new form because the template FMB you copied contains references to those objects. Changing the master files will change the references in all forms built from the template.

The methods for implementing template systems change with each new technology. Now that we Oracle developers are using or are considering using JDeveloper and Java Enterprise Edition (Java EE, formerly J2EE) technologies to create applications, we need to know how templating works in this new environment.

This white paper explores the subject of template use in JDeveloper 10.1.3.x Java web applications. It starts by discussing basic strategies for using templates in Java EE applications. Oracle is using the ADF Faces component set to create Fusion Applications—a rewrite of the E-Business Suite (Oracle Applications)—because ADF Faces has many productivity and usability benefits for development of Java EE web applications. Therefore, this paper focuses on techniques for creating templating concepts into ADF Faces applications. It also mentions additional Java frameworks used for templating such as Tiles, Facelets, and Velocity as well as how the JDeveloper plug-in, JHeadstart 10.1.3, uses templates.

In addition, the paper describes the ADF Faces skinning feature and the default skin Oracle Browser Look and Feel (BLAF) and provides tips about how to change the look and feel skin of an application. Finally, the paper mentions the additional features we can expect to see when using templates in JDeveloper 11*g* based on the preview version available at this writing.

Using Templates for a Java EE Web Application

When discussing user interface development, the term *template* refers to a file that contains elements common to many or all pages in the application. In the Oracle Forms example mentioned before, a template is an FMB file containing Forms objects common to each form as well as references to common building block objects in object libraries, PL/SQL libraries, or other forms. Some elements in the Forms template are copied and some are referenced.

Templates in Java EE applications follow the same general strategies as templates in Forms applications. You create a template file that you will copy as a basis of each new page you create. This template file contains common elements that are dynamically included from other files (like the objects in the Forms template FMB that are referenced from the object library). The common element files can then be changed when a change to the common look and feel is required. Once you have created the common element files and dynamically included them in the template file, you follow these steps in JDeveloper to create a page file:

1. Select the template file in the JDeveloper Navigator.
2. Select **File | Save As** from the menu. Navigate to the desired directory for the new page file.

3. Type in the new name of the file and click Save.

This process results in a complete copy of the template file that you can use for the new page. In the future, when a change is needed to a common element, you just change that element and all pages created from the template that uses that element will be automatically updated.

An Example

Consider the page layout shown in Figure 1.

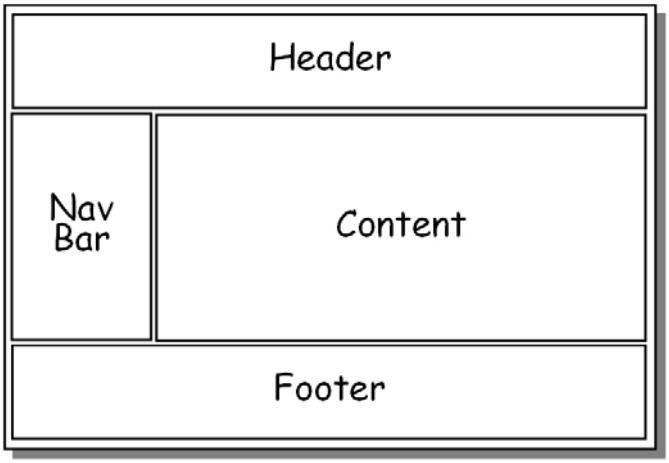

Figure 1. Sample page layout

In this design, the Header, Navigation Bar (Nav Bar), and Footer sections contain common elements that appear on each page of the application. The Content section is different for each page. In the template, this section contains a placeholder container element that will hold components specific to the functionality of the page.

You can set up a template to implement this design in two ways:

- A single common elements file
- Multiple common element files

> **Note**
>
> If you've worked with pages that use HTML frames, the idea of "includes" will sound familiar. HTML frames are built from separate HTML files just as a page with includes incorporates other pages at runtime. However, older browsers may not support frames. Also a page displayed with frames is not treated as a single page for purposes such as page submits and printing whereas a page with includes is treated as a single page. Therefore, the general industry preference is to avoid the use of frames.

Single Common Elements File Approach

In this approach, using the sample design from Figure 1, you create a single JSP template file containing the main layout (table with 4 cells) and all common elements. You also create separate pages containing only the content for each distinct page in the application. The template includes a dynamic "include" tag for its content cell. When the application is run, the Controller determines which content page will be used based on page flow logic. This approach is depicted in Figure 2.

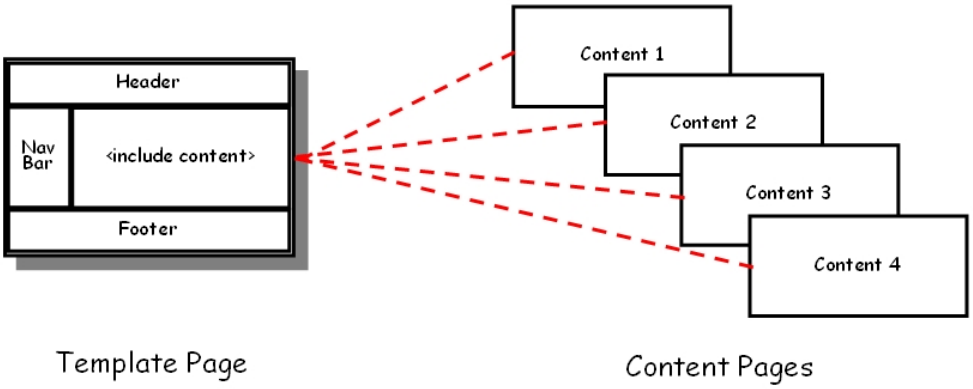

Figure 2. Using a single common elements file

The advantage of this approach is that you do not need to repeat the 4-cell layout for each page in the application. This makes changing the entire layout of the application much easier because you only need to change a single template file to affect layout on all pages in the application. The content pages are devoid of any common layout elements although they will contain layout elements required for specific content. Developing a new content page requires no work to implement the common layout design although you may not be able to run the content page without the template page (for example, for testing purposes).

The disadvantage of this approach is that it is a non-standard use of the Controller and requires custom Controller-level code to implement page flow.

Multiple Common Element Files Approach

In this approach, you create separate template JSP pages for each common element area. For the design in Figure 1, you would create a page for the Header, a page for the Nav Bar, and a page for the Footer. As with the Single Common Elements File Approach, you also create separate pages for each distinct content need. In this strategy, however, the layout (4-cell arrangement in this example) is coded into each of the content pages; each page contains “include” tags for all common element pages. The common element template pages have no overall (4-cell) layout. When the application is run, the template pages are incorporated automatically into each content-oriented page as shown in Figure 3.

The advantage of this approach is the same as the Single Common Elements File Approach: the common template areas are shared by all content pages. The main difference with this approach is that the content pages hold the common layout. Another advantage of this approach is that no special Controller code is required. Standard Controller code manages page flow and “include” tags combine the template pages with the content pages at runtime.

The disadvantage of this approach is that although the individual template pages define the common elements, the page layout for the common elements is repeated on each content page. As a result, if the layout is changed after a number of pages are created, the layout of all content pages will need to be modified. Although this scenario may not be very frequent, it may be time consuming with an application that is made of a large number of pages. However, the task of updating each page in this scenario is often just a matter of copying and pasting element nodes into the Structure window in JDeveloper. Moreover, if the application includes a large number of pages, you can build macros or Ant scripts to automate the restructuring.

Since this approach uses the Controller in a more standard way, it is the usual choice when selecting a method for template use. Therefore, the examples in the rest of this paper focus on this approach.

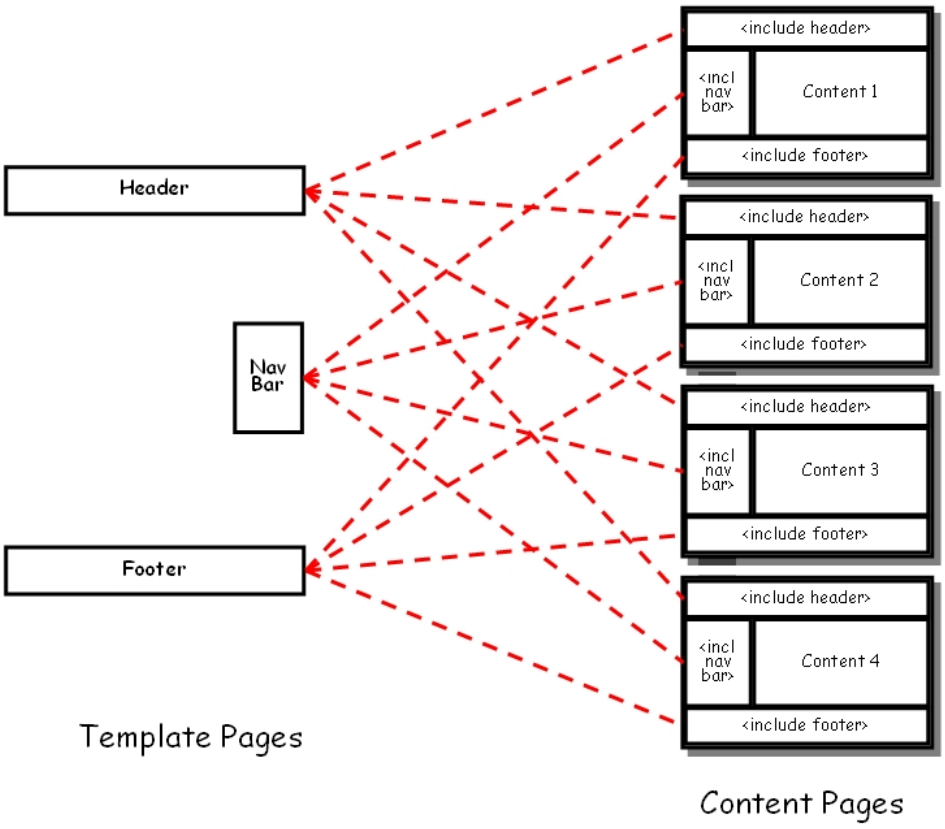

Figure 3. Using multiple common element files

Templates in Java-based Web Applications

Although developing applications using templates is a generally-recognized best practice, the Java EE standards do not yet mention templates. The usual flow of technology in the Java world is from the developer community level. After the community develops solutions to common problems and uses them for some time, the Java EE standards are eventually updated to include a standard solution. The community and Sun Microsystems work on the standard and, after it is ratified and documented in the Java EE standards, the community it as the best way to accomplish a task.

The use of templates is at the developer community stage of this process. Many frameworks have been developed, used, and refined in an attempt to find the best method for creating UIs based on templates. All this work is ongoing but currently has shaken down into these popular template frameworks:

- Tiles
- Facelets
- Velocity

Although you can use any of these frameworks while working in JDeveloper, the 10.1.3.x version of the tool has no native support for any of them. A very brief overview of each will help the context of the JDeveloper solutions in a later section.

Tiles

Home website: struts.apache.org/struts-tiles

Tiles is an open source initiative in the Struts framework. (Apache Struts is known best for its very popular Controller framework.) Tiles currently requires the use of Struts as the Controller framework in the application but a standalone version of Tiles ("Tiles2" or "Standalone Tiles") is under development.

If you are creating JavaServer Faces (JSF) code and are using the native Controller, you are probably not using Struts and the use of Tiles for templates is probably not indicated.

Tiles and JDeveloper ADF

Tiles support is not native to JDeveloper but an article on OTN explains how to use Tiles with ADF Faces applications (www.oracle.com/technology/pub/articles/ vohra_tiles.html). In addition, a general Tiles tutorial appears here: www.jsftutorials.net/tiles/jsf-tiles.html.

Facelets

Home website: facelets.dev.java.net

Facelets is another open source initiative that is considered as the framework that will be included in the future Java EE standards. Facelets covers work in areas other than templates, but it has good support for all template concepts. It is a strong contender for templating in any JSF application that might need Java EE standard support in the future.

Facelets and JDeveloper ADF

Support for Facelets is not native to JDeveloper but information about how to use JDeveloper with Facelets appears on the Facelets home page mentioned before. Other introductory articles appear here: www.jsfcentral.com/articles/facelets_1.html and http://www-128.ibm.com/developerworks/java/library/j-facelets/

Velocity

Home website: velocity.apache.org

Velocity (Apache Velocity Engine) is another Apache templating framework project. Facelets can be used for development of user interfaces other than web pages. It offers a scripting language that contains conditional and iteration statements and references to variable values. It enforces Model-View-Controller (MVC) design, which allows different developers to work on different layers of the application at the same time without overlapping each other's code.

Velocity and JDeveloper ADF

JDeveloper supports work with Velocity in the same way it offers support for work in any Java technology. JDeveloper does not have special editor or wizard support, but, as with any Java technology, you can plug in the Velocity framework libraries and edit source code with Velocity elements using the code editor.

Velocity and JHeadstart

More notable than the Velocity support in JDeveloper is the use of Velocity in JHeadstart, a JDeveloper plug-in created and maintained by Oracle Consulting's Center of Excellence in the Netherlands. (Search for "JHeadstart Product Center" in Google to find the JHeadstart home page.)

Once you define the application declaratively using the JHeadstart Application Definition Editor (in a way that is very similar to the declarative property screens in Oracle Designer), you run the JHeadstart Application Generator (JAG) to create JSF pages and Controller code. JAG uses Velocity templates as a basis for the code it creates. JHeadstart (an extra-cost product) ships with a complete set of templates to generate any ADF Faces JSF object including tables, forms, trees, shuttles, tab menus, and Controller code. However, you can create your own templates for any or all objects and declare that JAG uses them instead of the shipped templates.

For example, you can define a special template for a button and another template for the page layout. Used in this way, templates in JHeadstart work similarly to a Forms object library for the Oracle Designer Forms Generator. The object library in this case provides a look and feel for basic items or for window- or canvas-level components. In Forms work outside of Designer, templates for the JHeadstart generator are like object library items that are copied to a new form.

JAG creates objects for the UI based on declared properties in the application definition file. After reading the item's properties, JAG then reads the appropriate template and based on Velocity scripting logic and ADF Faces tags in the template, it generates the code for that object into the page file. A similar process occurs for Controller code. Although JHeadstart does not use Velocity for runtime includes of common object files, it does make good use of the ADF Faces template component, af:region (described in a later section).

ADF Faces Skins

When developing a template system, you need to start with the design for a common look and feel. This type of design requires thinking about many different elements that will appear on the pages. After the design is created, you will need to develop style sheets that implement shared color and font attributes for the common elements and areas on the pages.

To assist in applying your look-and-feel design, you can take advantage of the ADF Faces *skins* feature. Skins offer the ability to change the entire appearance of an application by assigning a single configuration property to point to a different skin definition. A skin definition consists of cascading style sheets (CSS) that define the visual attributes as well as a resource bundle that stores text that appears inside the components.

The configuration property that identifies the skin definition is located in the adf-faces-config.xml file (in the WEB-INF directory) of an ADF Faces application. To access this property in the Property Inspector of JDeveloper, select adf-faces-config.xml in the Navigator, and select the *skin-family* property in the Structure window as shown in Figure 4. Alternatively, as always, you can just open the code file and edit the XML directly.

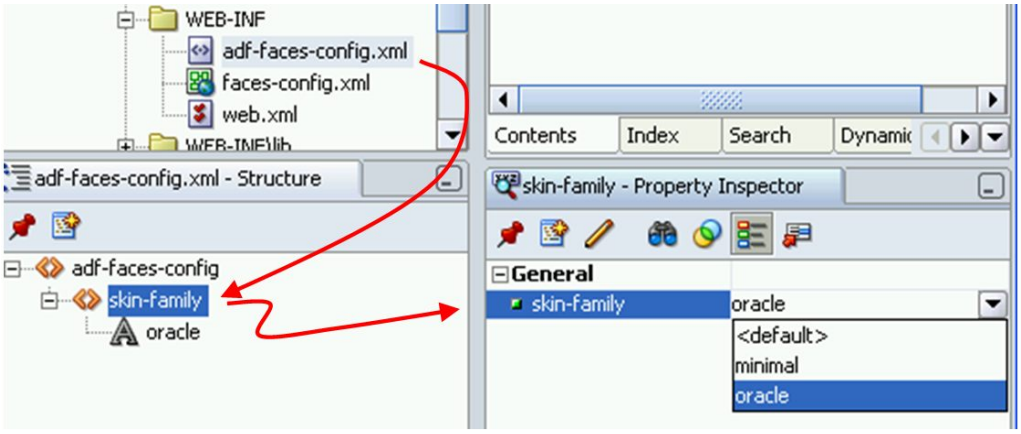

Figure 4. Setting the skin for an ADF Faces application

JDeveloper ships with three skins: oracle, minimal, and simple. The *oracle* skin implements the Browser Look and Feel (described in a later section), a fully defined skin backed by extensive documentation. For example, the following shows ADF Faces navigation buttons and menu tabs in the oracle skin:

The *minimal* skin contains few non-default elements. The same buttons and menu tabs would appear as follows in the minimal skin:

The *simple* skin is actually no skin at all and applying it is the same as misspelling a skin name as shown in the following screenshot. Notice that the menu tab decoration has been omitted and the tabs are displayed as highlighted links.

You can create code to switch skins at runtime so you can offer users the ability to switch their look and feel as desired. This technique may not be useful in your application, but it emphasizes the ease with which a skin can be changed.

Why Use Skins?

Skins fit well with fundamental template concepts. If your template file uses a skin, you will be able to radically change the look and feel of your application in the future by defining a new skin or by changing the existing skin and then changing the single configuration property of the application to cause all pages to use the new skin. You do not need to redefine or touch any common element files for this type of design refresh. Of course, if the new design requires changes in the locations of common elements, a simple skin change will not suffice.

> **Note**
>
> Skins can modify fonts, colors, element shapes, and anything else that can be defined in style sheets.

Oracle Browser Look and Feel (BLAF)

When creating a skin, it is helpful to take inspiration from a fully-developed look and feel. The Oracle Browser Look and Feel (BLAF), mentioned before as the oracle skin, fits this bill. BLAF is backed by an extensive set of standards for designing the appearance and workings of the user interface layer of an application. Oracle develops and maintains these standards primarily to support the E-Business Suite (EBS or Oracle Applications) modules that present web browser interfaces.

The detailed BLAF standards are available in hundreds of pages of web files (www.oracle.com/technology/tech/blaf). Although Oracle online applications apply the BLAF standards currently to ADF UIX user interface code, the standards are perfectly suitable to ADF Faces. In fact, ADF Faces uses BLAF standards for its default look and feel.

> **Note**
>
> Although EBS offers BLAF as the default skin, the skinning feature allows Oracle customers to use their own look and feel by changing skins. If your organization has performed this type of customization, your EBS applications likely use a look and feel specific to your organization instead of BLAF.

Creating Your Own Skin

If you decide not to use BLAF, you can develop your own skin. The process is detailed in many white papers referenced later, but it is useful to get a picture of the steps and effort involved. The first step is to prepare by reviewing details of the work, and to budget time for this work. Creating a skin entails working primarily with style sheets; the work is not difficult but it can be time-consuming if you want to be thorough. Another preliminary step is defining a complete graphical design that you wish to implement. It will not be productive to make extensive change to the design while coding the skin.

Next, you start with the simple skin and define property values for specially-named CSS selectors (styles). This process will override the default look and feel that ADF Faces components use. Working with the CSS selectors will take the bulk of your skin development time. The last development step is to register the skin in the adf-faces-skins.xml file so it is available to the *skin-family* property. You activate the new skin by changing that property as described before.

> **Tip**
>
> When designing a look and feel for your organization, you may find it helpful to review the BLAF documentation (www.oracle.com/technology/tech/blaf) to get ideas about what type of standards to create and how the standards for different elements must complement each other.

Resources for Skin Development

The following will get you started learning about and developing a custom skin:

- **Introduction**: JDeveloper help system topic *Selectors for Skinning ADF Faces Components*

- **More information:** *ADF Developer's Guide* Chapter 22 (section 22.3) available on otn.oracle.com and in JDeveloper help system
- **A sample skin with the OTN look and feel**: blogs.oracle.com/jheadstart/2006/12/22#a122
- **ADF Faces skin selector documentation**: www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange /jsf/doc/skin-selectors.html
- **Changing skins at runtime**: "ADF Faces 10.1.3: Setting Skins per User Role" by Olaf Heimburger; blogs.oracle.com/olaf/2007/04/23
- **White paper**: "Developing and Using ADF Faces Skins" by Jonas Jacobi; www.oracle.com/technology/products/jdev/ 101/howtos/adfskins/ index.html
- **Documentation**: *Oracle WebCenter Framework Developer's Guide* Chapter 9 "Defining and Applying Styles to Core Customizable Components"; download.oracle.com/docs/cd/B32110_01/webcenter.1013/b31074/toc.htm

Techniques for Creating Templates in JDeveloper

When creating the template in JDeveloper, you can use standard JSP or ADF Faces components to include common element files in your page, for example:

- **jsp:include**
- **af:region**

In addition, you can take advantage of the power of ADF Faces *container components*—components that hold other components.

ADF Faces Container Components

Using container components is a best practice when working with ADF Faces because they save time and contribute to look and feel consistency. These components are usually named with a prefix of "panel," for example, af:panelForm, af:panelBorder, af:panelHorizontal, and af:panelPage. (The "af" alias acts as shorthand for the library in which the tag is located.)

Container components are similar in concept to HTML tables, which hold other components in a row-column arrangement. In fact, if an ADF Faces container component is rendered for an HTML browser client, it will appear as an HTML table or a set of nested HTML tables. However ADF Faces container components are much easier to use. For example, you do not need to worry about adding spacer graphics, setting up table rows and columns, and setting padding and cell span properties as you do with HTML tables.

A container component offers special layout properties for maintaining relative positioning of its children when the browser window is resized. It also offers predefined layout areas called *facets*. Components held in facets will maintain their assigned position regardless of the window size. This kind of functionality is possible with pure HTML tags, too, but setting this up in HTML is much more time consuming and error-prone.

When you add a container component to a JSP page in JDeveloper, the Visual Editor will display some or all of these facets as shown for the following af:panelBorder component:

Notice that facets are displayed in the visual editor with dotted boxes surrounding the name of the facet. You can choose to add any element to any of the facets, or not use the facets at all. The rule is that only one component can be contained in a facet, but that one component could be another container component, which could contain many components.

Consider the sample template described in an earlier example. The af:panelBorder container provides facets that suit the Nav Bar (left facet), Header (top facet), and Footer (bottom facet) areas of this template. This tag would appear in the JSP template file as follows:

```
<af:panelBorder>
  <f:facet name="top">
    <!-- Header -->
  </f:facet>
  <f:facet name="bottom">
    <!-- Footer -->
  </f:facet>
  <f:facet name="left">
    <!-- Nav Bar -->
  </f:facet>
  <!-- Content -->
</af:panelBorder>
```

In this code snippet, the comment lines would be replaced with other elements. In the case of the template, the Header, Footer, and Nav Bar comments would be references to common element files that contained the relevant components. The Content comment would be replaced by whatever the page was intended to display. Common elements files can be referenced using the jsp:include or af:region tags.

> **Note**
>
> You do not need to use all facet areas and that content can appear outside of facets in a container component. The content outside the facets will be positioned in the available free space—in this case, the center and right part of the page.

jsp:include

The jsp:include tag is a standard JavaServer Pages (JSP) tag that is available in the JSP page of the JDeveloper Component Palette. You can use jsp:include to embed one page inside another at runtime. For example, you can include the following in your JSP:

```
<jsp:include page="leftNavBar.jsp"/>
```

This code snippet will embed the contents of the leftNavBar.jsp inside the existing page at whatever point the jsp:include appears. If the tag is contained as follows, the contents of leftNavBar.jsp will be displayed in the left facet of the af:panelBorder container:

```
<af:panelBorder>
  <f:facet name="left">
    <f:subview id="navBar">
      <jsp:include page="leftNavBar.jsp"/>
    </f:subview>
  </f:facet>
</af:panelBorder>
```

In this example, the f:subview tag surrounds the jsp:include tag as required when using jsp:include in a JSF page.

> **Note**
>
> JDeveloper will display the contents of the included page in the Visual Editor but you need to open the page file that is included to edit those contents.

af:region

The af:region tag is an ADF Faces container component. It is not available in the Component Palette by default, but you can either type the tag into the Code Editor or add it to a custom page in the Component Palette and then drop it to the screen from that page. Since the tag is not visual and is not represented in the Visual Editor, typing the tag is probably the most appropriate option. This tag is used to include one page in another just as with the jsp:include tag.

This tag requires a small bit of indirection. That is, after creating a common element file for the contents that you want to include, you refer to that file in a configuration file, region-metadata.xml. This XML file contains a list of each file included by af:region tags and assigns the included file a fully-qualified name. This fully-qualified name is then used inside the af:region tag in the main JSF file. Creating a region consists of three main steps:

1. Create a Region Definition

Content in the common elements file must be enclosed in an af:regionDef tag to signify that they will be defining region content. Using the sample template described earlier, you would define a common elements JSF file (topMargin.jsp) with the following contents:

```
<?xml version='1.0' encoding="windows-1252"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:af="http://xmlns.oracle.com/adf/faces"
          xmlns:afh="http://xmlns.oracle.com/adf/faces/html">
  <jsp:output omit-xml-declaration="true" doctype-root-element="HTML"
              doctype-system="http://www.w3.org/TR/html4/loose.dtd"
              doctype-public="-//W3C//DTD HTML 4.01 Transitional//EN"/>
  <jsp:directive.page contentType="text/html;charset=windows-1252"/>
  <af:regionDef var="attrs">
    <af:objectImage source="/images/header.jpg"/>
  </af:regionDef>
</jsp:root>
```

In this file, the xml, jsp:root, jsp:output, and jsp:directive tags are standard elements that set up the JSF page. The af:regionDef tag encloses the header graphic file defined by the af:objectImage tag. (This is a very simple header. Normally, web page headers contain links and other layout elements.)

2. Set up the Region Metadata

You then enter the following element in the region-metadata.xml file (located in the META-INF directory):

```
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  <component>
    <component-type>hrapp.view.region.topMargin</component-type>
    <component-class>oracle.adf.view.faces.component.UIXRegion</component-class>
    <component-extension>
      <region-jsp-ui-def>/regions/topMargin.jsp</region-jsp-ui-def>
    </component-extension>
  </component>
</faces-config>
```

The faces-config element identifies the type of XML file. The component element code defines the region JSP file (in the region-jsp-ui-def tag nested in the component-extension tag) and names it as a region type (in the component-type tag). In this case, the region definition name is “hrapp.view.region.topMargin.” The component-class element declares the ADF Faces class used to implement the included region (UIXRegion). This class is always the same for ADF Faces applications.

3. Reference the Region in the JSF File

The last step is to reference the region in your JSF file. In the template example, the header region would be used in code like the following. (This file would also contain all normal ADF Faces container tags such as f:view, afh:html, afh:body.)

```
<af:panelBorder>
  <f:facet name="top">
    <f:subview id="topMargin">
      <af:region id="topMargin"
                 regionType="hrapp.view.region.topMargin"/>
    </f:subview>
  </f:facet>
</af:panelBorder>
```

As with the jsp:include, the f:subview container surrounds the af:region tag. The af:region tag is assigned an *id* property. This is handy in case you need to reference it in code. It also is assigned the *regionType* property, which references the fully-qualified region definition name you set up in step 2.

Which Tag to Use?

If you are using ADF Faces components to build your JSF files, it makes the most sense to use af:region because it is a JSF component and you can write backing bean code in Java to modify its handling. It is also an ADF Faces component with more properties and more chance for enhancements (because it is more modern than jsp:include).

If your JSF files do not use ADF Faces, the jsp:include tag is probably the best option.

> **Note**
>
> If you use the JSP Standard Tag Library (JSTL) library tags in your application, you alternatively might want to take advantage of the JSTL tag c:import, which works like jsp:include. Although c:import has more functionality than jsp:include, if your application uses ADF Faces, it is still better to rely on af:region to embed common content in a template.

Templating in JDeveloper 11*g*

Although this white paper focuses on alternatives and strategies for using templates in ADF within JDeveloper 10.1.3.x, you may be wondering how this will change with the next release, JDeveloper 11*g*. Oracle has released a preview version of 11*g* on otn.oracle.com, so you can examine its enhanced features for creating and using templates. You will find that, although there is more built-in support for templates in JDeveloper 11*g*, the template concepts are not much different from JDeveloper 10.1.3.x. Moreover, getting started with templates in JDeveloper 10.1.3 will allow you to more easily assimilate and appreciate the enhancements for template support in JDeveloper 11*g*.

Building the Template

The New Gallery in JDeveloper 11*g* contains an option to create a template JSF file. During the template creation process, you define and name *facet definitions* into which you will place certain content when you use the template. For example, the left side of all pages in your application might contain a navigation bar and the right side might show the main page content. You set up these facet definitions and name them "navControl" and "mainContent," respectively. When the template is used, the facet names will appear to the intention for those areas just as with the ADF Faces container components in JDeveloper 10.1.3.

In addition, you can define parameters that will hold values passed to the template from the page. You can also specify that a PageDef (binding) file be associated with the template. This PageDef file is useful if the template will hold data-aware components.

This template idea is similar to the prebuilt ADF Faces container components (such as af:panelBorder) in JDeveloper 10.1.3. However, in JDeveloper 11*g* ADF Faces, you define your own facets, instead of relying on the predefined facets available for a particular container component.

Using the Template

When you create a JSF file, the Create JSF Wizard will offer a list of the available templates in the application. Selecting a template will reference all template elements with the tag af:pageTemplate. For example, the template just described (called "navContent.jspx") would be referenced in a new JSF page with the following code:

```
<af:pageTemplate viewed="/navContent.jspx">
  <f:facet name="navControl" />
  <f:facet name="mainContent" />
</af:pageTemplate >
```

In this code snippet, the two facets were defined and laid out in the template creation process. Their functionality and components are held in the template. This simplifies the development of pages that require similar look and feel aspects because the common elements are referenced in the page from the template file. In this way, the JDeveloper 11*g* template functions use the approach of a Single Common Elements File described earlier in this white paper.

If you defined a template attribute and referred to its name in a template component, you can assign a value to that component by coding an f:attribute tag in the JSF page with the name of the attribute and the value you need to transfer to the

common element. For example, an af:outputText component in the template (navContent.jspx) could display a page title using this tag:

```
<af:outputText value="#{attrs.PageTitle}" />
```

The JSF page you are building would then assign a value to this component in the template by using this tag inside the af:pageTemplate tag shown earlier:

```
<f:attribute name="PageTitle" value="This is a Sample Page Title" />
```

> **Note**
>
> A brief video that demonstrates template creation and use in JDeveloper 11g is available at www.oracle.com/technology /products/jdev/11/index.html.

Conclusion

If you are using ADF Faces to build your applications, it is best to rely on the af:region tag when building your template. It is also useful to consider defining a skin that supplies common look and feel aspects. If you are not using ADF Faces, you will likely rely on plugging Facelets into JDeveloper or on using standard tags such as jsp:include or c:import. Whichever path you chose, you can rely on knowing that planning and implementing a strategy for template use will save time and effort in the long run and further your ability to provide users with a consistent look and feel in the application.

###

Peter Koletzke is a technical director and principal instructor for the Enterprise e-Commerce Solutions practice at Quovera, in Mountain View, California, and has 24 years of industry experience. Peter has presented at various Oracle users group conferences more than 200 times and has won awards such as Pinnacle Publishing's Technical Achievement, Oracle Development Tools Users Group (ODTUG) Editor's Choice, ECO/SEOUC Oracle Designer Award, ODTUG Volunteer of the Year, and NYOUG Editor’s Choice. He is an Oracle Certified Master, Oracle Ace Director, and coauthor of the Oracle Press Books: *Oracle JDeveloper 10*g *for Forms & PL/SQL Developers* (with Duncan Mills); *Oracle JDeveloper 10*g *Handbook* and *Oracle9*i *JDeveloper Handbook* (with Dr. Paul Dorsey and Avrom Roy-Faderman); *Oracle JDeveloper 3 Handbook*, *Oracle Developer Advanced Forms and Reports*, *Oracle Designer Handbook*, *2nd Edition*, and *Oracle Designer/2000 Handbook* (all with Dr. Paul Dorsey).