

... AND UIX BEGAT ADF FACES: HOW RICH IS ADF FACES RICH CLIENT?

Peter Koletzke, Quovera

- ¹ In the beginning Oracle created E-Business Suite (EBS).
- ² And E-Business Suite was with Forms but void of lightweight client.
- ³ And heaviness was upon the face of the browser.
- ⁴ And Oracle said, "Let there be *User Interface XML* (UIX)": and there was UIX.
- ⁵ And Oracle used UIX to build E-Business Suite.
- ⁶ And the JavaServer Faces (JSF) standard divided UIX from industry-accepted development.
- ⁷ And Oracle said unto its Development Wing, "Make UIX fruitful and multiply it."
- ⁸ And the Development Wing brought forth JSF-standard components and increased them in number.
- ⁹ And Oracle called it "ADF Faces." Thus UIX begat ADF Faces.
- ¹⁰ And it was good.
- ¹¹ And ADF Faces begat Trinidad; and ADF Faces also begat ADF Faces Rich Client.
- ¹² And Oracle said, "Let ADF Faces Rich Client have dominion over the Fusion Applications."
- ¹³ And Oracle said unto its customers, "Behold, we have given you ADF Faces Rich Client; to you it shall be a tool."
- ¹⁴ And, behold, it was very good.

This white paper explains the features of the Application Development Framework (ADF) framework ADF Faces Rich Client (ADF Faces RC). It demonstrates basic functionality and standard techniques for working with some of the ADF Faces RC components. In answering the question posed in the white paper's title and so— interpreting *rich* in this context as "highly-interactive," "flexible," and "easy to use"—this handful of questions arises:

- **What is ADF Faces and where did it come from?**
- **How does partial page rendering work?**
- **How can I achieve the perfect layout?**
- **What kinds of atomic components are available?**
- **How can I change the default ADF Faces appearance?**
- **How will I ever learn about all these components?**

Answering these questions should give you a picture of some of the features of ADF Faces RC so you can come to a conclusion about how rich (highly-interactive, flexible, and easy to use) it really is. Although descriptions and illustrations can help in understanding the answers to these questions, another key method for understanding is "doing." Therefore, the white paper also provides some short, hands-on practices so you can try out in JDeveloper 11g to get tastes of ADF Faces features and components.

Note

The screenshots and practices in this white paper were developed with JDeveloper, version 11.1.1.1.0 (build 5407). If you use a later release, the exact appearance may be different and you may need to adjust steps to changes introduced with that later release. However, the principles and steps should remain the same regardless of the version.

We didn't need dialogue.

We had faces.

—Norma Desmond, *Sunset Boulevard* (1950)

What Is ADF Faces and Where Did It Come From?

ADF Faces is a JSF component set that provides a group of rich user interface controls for developing JSP pages with JSF components. ADF Faces is well integrated with JDeveloper, and the developer’s experience using ADF Faces can be mostly declarative and visual. In addition, as mentioned before, ADF Faces is one of the technologies Oracle uses to build the Fusion Applications.

ADF Faces consists of class libraries and tag libraries of several categories. This white paper focuses on these two main categories that you will use to build JSF pages:

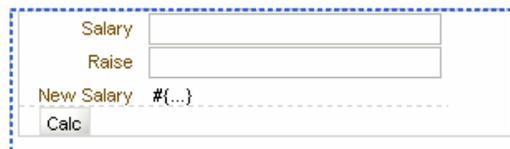
- **Atomic components**, which display a single object such as a text field with its prompt (`af:inputText`) or a pulldown (`af:selectOneChoice`).
- **Layout containers**, which surround atomic components and other layout containers and offer automatic layout features such as field alignment (`af:panelFormLayout`).

One of the last runtime processing steps for a JSF page is to pass the components through a *render kit*, a translation layer that converts the raw representation of the components into markup tags specific to the client display device.

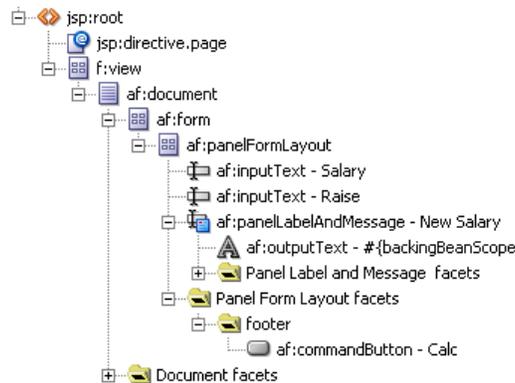
HTML Render Kit

When the target client device is a web browser, ADF Faces uses the HTML render kit to form the component hierarchy into HTML tags that the web browser can display. Therefore for a web browser target device, an ADF Faces component such as `af:panelGroupLayout` will create an HTML table structure using tags such as `<table>` and `<tr>`. An ADF Faces text field component such as `af:inputText` will render as an HTML input tag, for example, `<input type="text" />`.

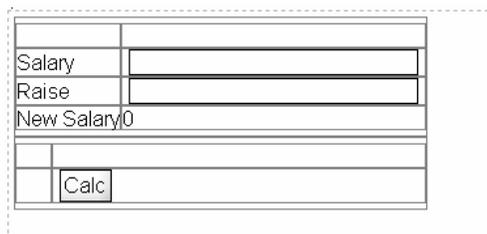
As a visual example of this translation, the first hands-on practice in this white paper creates the small form shown here in the JDeveloper visual editor:



The tag hierarchy that represents this form is shown next in the Structure window:



At runtime, the web page is created with standard HTML tags that would display in the JDeveloper visual editor as shown next. Notice the use of nested HTML table elements.



Note

ADF Faces tags use the “af” prefix to refer to the tag library. The tags are represented with a friendlier name in the Component Palette. For example, the `af:inputText` tag is labeled as “Input Text” in the Component Palette. This white paper uses both types of names interchangeably.

A Bit More on the History Side

Oracle E-Business Suite (EBS) applications were first developed using Oracle Forms. Although currently Oracle Forms applications are run in a rich-client, web browser environment, Oracle saw the need for a lightweight, Java EE–oriented runtime environment and met the need in subsequent versions of EBS (up through and including R12) by developing and using *UIX* (formerly an expansion for “User Interface XML”).

UIX is a framework consisting of components used to build user interfaces. *UIX* files are coded in XML (as the name implies), and the files run in a standard Java EE web client environment. *UIX* files are interpreted by the *UIX* Servlet process running in a standard Java EE web container. The servlet constructs an HTML page (for web browser clients) and that page returns to the client browser. *JDeveloper 9i* was the first release to offer development tools support for *UIX*; with the introduction of ADF in *JDeveloper 10g*, the framework was renamed to ADF *UIX* and support in *JDeveloper* continued through release 2 (10.1.2). This support was set aside in *JDeveloper 10.1.3* to make way for JSF support.

The major benefit of *UIX* over other HTML-rendered alternatives available at the time was its high interactivity features. Unlike component sets that preceded it, *UIX* components in a web application could rapidly return feedback to the user in a way rivaling thick client applications. This type of rich, highly-interactive interface is now known as the main user-facing aspect of Web 2.0.

After Oracle had released EBS applications that successfully used *UIX*, the industry adopted JSF technology as a Java EE standard. Oracle also rewrote the proprietary ADF *UIX* tag library to be completely compliant with the JSF standard and called this new framework “ADF Faces.” *ADF Faces*, therefore, is a JSF-compliant UI component set that is based on the JSF API. It is most commonly used to output HTML for web browsers, but it is capable of output to any type of display device, with native support for wireless devices. This multi-faced output style is defined in the JSF standards but is rarely implemented.

In *JDeveloper 10g*, Release 3 (10.1.3), tools support for ADF *UIX* was phased out in favor of support for ADF Faces 10g. *JDeveloper 11g* introduces ADF Faces Rich Client (ADF Faces RC, referred to in this white paper as just “ADF Faces”) as a further evolution on and improvement of ADF Faces 10g.

Trinidad

Early in 2006, Oracle donated ADF Faces 10g to the open-source Apache MyFaces Project (myfaces.apache.org). Although Oracle continued to enhance and support ADF Faces for its customers, the larger Java community also contributed to the functionality of the donated version. The part of the MyFaces project based on ADF Faces 10g is called *Trinidad*. ADF Faces RC is a separate code base, which is at the moment maintained only by Oracle. The *Trinidad* library is also supported in *JDeveloper 11g*, and ADF Faces RC is built on top of it.

ADF Faces Is Better

Since “better” is a relative word, the first question is, “What is ADF Faces better than?” The short answer is that it is the latest step in an evolution, and therefore better than any of its predecessors (*UIX*, ADF *UIX*, and ADF Faces 10g). ADF Faces is arguably better than other JSF component sets because of its long-term evolution (few competing frameworks have such a long heritage) and its native features that make declarative and visual development of highly-functional user interfaces faster and easier. To further answer the question, it is helpful to know more about its features, which will be explored in this white. One such feature is partial page rendering.

How Does Partial Page Rendering Work?

Of all ADF features, partial page rendering (PPR) is one of the most powerful because it offers the developer a declarative way to create highly-interactive web pages. PPR uses JavaScript and XML to submit and update only some fields or data on

a page. The concept behind PPR (and the use of JavaScript in general) is to achieve a more user-friendly interface by mitigating or completely removing the wait a user experiences when a full page is submitted to the server and re-rendered after processing occurs.

You are no doubt familiar with web pages that rely on the full-page submit technique. You also may share the frustration that your experience with an application or website is interrupted by waiting for a server to return and redraw a web page. One of the tenets of Web 2.0 is that user interfaces are highly interactive and wait times as well as the ensuing frustrations are reduced.

PPR was one of the strengths of the early releases of ADF UIX and provided this high interactivity to Oracle EBS applications. Subsequently, the Java community evolved the core ideas behind PPR, and the technique that emerged is now called “Asynchronous JavaScript and XML” (AJAX). *AJAX* outlines a strategy for combining these two technologies to provide a user interface that responds faster than pages built without AJAX.

ADF Faces offers the PPR capability but works within the guidelines of the popular industry technology AJAX. Unlike typical AJAX development, where you need to write custom JavaScript code, define JavaScript events, and create custom Java servlet code to handle the requests, using PPR in ADF Faces is completely declarative. It could even be called “declarative AJAX.”

You can think of PPR in simple terms as a process outside the normal page submit and full page rendering whereby a source component’s change in value (or a user event such as a button click) triggers a target component to be re-rendered. As the target component is re-rendered, its property values are read and, in the case of EL expressions, reevaluated. The source component needs to be marked so that it automatically submits its value for processing; the target component needs to be marked so that it watches (listens) for a partial submit from one or more source components. You use the following properties to mark the source and target components:

- **AutoSubmit** Set this property to “true” on all source components that will initiate the partial page submission.
- **PartialTriggers** Set this property on the target component or components to listen for a change in the source component. The value of this property is a space-delimited list of IDs for the components that will trigger the PPR re-rendering of the target component.

In addition, the *PartialSubmit* property of command components, such as `af:commandButton` and `af:commandLink`, declares whether that command component will initiate a PPR event. For example, you could declare a button and set its *PartialSubmit* property to “true” so that when the user clicks the button, any components declared with *AutoSubmit* as “true” will be re-rendered. You can also use components called *listeners* to trigger partial submits on non-standard component actions, such as right-click, selection, or rollover.

You might want a taste of PPR concepts now. If so, follow these (abbreviated) steps in *JDeveloper 11g*.

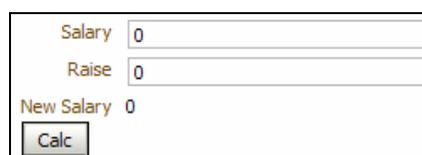
Note

As an additional reminder, you need to run ADF Faces RC applications in a compatible browser, for example, Internet Explorer 7.0 or later, Mozilla Firefox 2.0 or later, or Safari 3.0.

Hands-on Practice: Demonstrate Basic PPR

This hands-on practice shows how you can declare AJAX (PPR) functionality using the ADF Faces components. (The practices in this white paper assume you have a familiarity with basic operations in *JDeveloper 11g* such as creating an application.) The application you will build is a three-field form with a Calc button and Salary, Raise, and New Salary fields. The New Salary field will automatically update when the value of the Salary or Raise field changes.

You will be building a form that appears as follows at runtime:



The screenshot shows a web form with three input fields and a button. The first field is labeled "Salary" and contains the value "0". The second field is labeled "Raise" and contains the value "0". The third field is labeled "New Salary" and contains the value "0". Below the fields is a button labeled "Calc".

For simplicity, this practice uses a backing bean (managed bean) to provide a source for binding the user interface fields.

1. If any applications are running, stop them as well as the Default Server (using the Terminate button).
2. Create an application called “ADFFacesTest” using the “Fusion Web Application (ADF)” application template. Specify an application prefix of “demo.” This template will create two projects, although you will not use the Model project in this practice.

Caution

Remember to avoid typing the ending period or comma for property values we enclose inside quotes unless otherwise specified.

3. In the ViewController project, create a JSF page as an XML document with the name “testPPR.jspx.” (Start this process from the New Gallery.) In the Page Implementation section of the Create JSF Page dialog, specify that the components will be exposed in a new managed bean.

Additional Information: Automatically exposing UI components in a managed bean is something you will rarely do in a Fusion application, but we show it here to demonstrate how you can interact with managed beans using EL.

4. From the Component Palette, drop a Panel Form Layout component (from the Layout panel) onto the page.

Additional Information: The `af:panelFormLayout` component provides automatic alignment for fields it contains. You will drop in some fields next.

5. Drop an Input Text component (Common Components) onto the `af:panelFormLayout` node in the Structure window. In the Property Inspector, set *Label* as “Salary” and *Value* as “0.”

Additional Information: The *Value* property supplies a default value that will be overridden by input from the user.

6. Drop another Input Text component onto the `af:panelFormLayout` and set *Label* as “Raise” and *Value* as “0.”

Tip

Dropping a component onto a layout container such as `af:panelFormLayout` adds it under the other components within that container. You can reorder components within a container most easily using drag-and-drop operations in the Structure window.

7. Drop an Output Text component onto the `af:panelFormLayout`.

Additional Information: This read-only field will show the results of the calculation.

8. For the *Value* property, click the down arrow next to the field and select Expression Builder. Remove the existing expression. Enter or select nodes in the Expression Builder navigator to form the following expression (all on one line):

```
#{backingBeanScope.backing_testPPR.it1.value +
  backingBeanScope.backing_testPPR.it2.value}
```

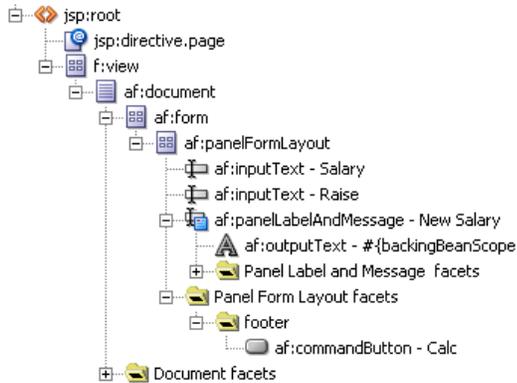
Additional Information: Open nodes under ADF Managed Beans\backingBeanScope to find the inputText (“it”) items. The name of your backing bean and text item IDs may vary a bit. This expression adds the values of the two fields and assigns the value to the Output Text item after the page is submitted.

9. Now you need to add a label for the read-only field. Notice that the `af:panelFormLayout` container left aligns the `af:outputText` field with the existing fields, but `af:outputText` has no *Label* property so the line looks incomplete. Fortunately, there is an easy solution—the Panel Label and Message component.
10. On the `af:outputText` component in the visual editor or Structure window, select **Surround With** from the right-click menu. Select “Panel Label and Message” and click OK. Set *Label* for `af:panelLabelAndMessage` as “New Salary.”

Additional Information: This embeds the `af:outputText` component within the `af:PanelLabelAndMessage` component, which supplies the label. Now the fields and labels line up properly.

11. Drop a Button onto the footer facet of the `af:panelFormLayout`. Set *Text* as “Calc.”

Additional Information: This button will submit the values to the server (Model layer of code, which for this application is included with the View Controller code). The addition specified in the New Salary *Value* expression will fill in when the user clicks this button and the page is refreshed with updated values from the server. The Structure window should display the following nodes:



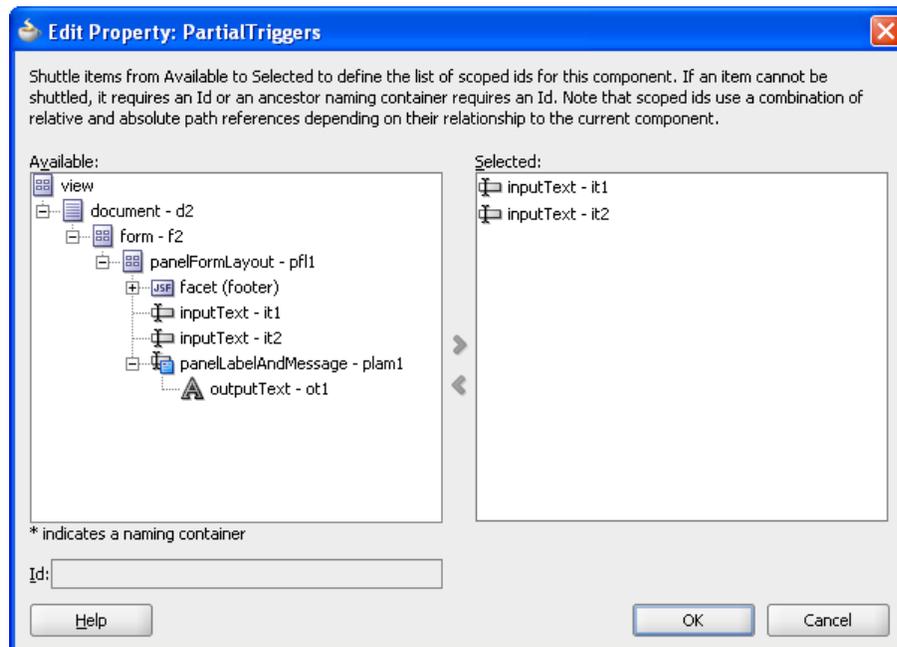
12. Run the page. Fill in Salary as 1000 and Raise as “200”; press TAB. The New Salary field value will not change. Click Calc. The New Salary field will display “1,200.”

Additional Information: This is the default behavior of a JSF page. No processing will occur unless the user actively submits the page, in this case, using a submit button. Now it is time to set up PPR so that the New Salary value will be refreshed automatically.

13. Close the browser.

14. On both Salary and Raise components, set *AutoSubmit* (Behavior section) as “true.” This will trigger PPR when either of the component values change.

15. On the New Salary `af:panelLabelAndMessage` component, click the Edit selection in the pulldown to the right of *PartialTriggers* and move the names of the other two items to the *Selected* area as shown here:



16. Click OK and the property will fill in with the names of the items separated by a space.

17. Click Save All. Run the page. Fill in Salary as “1000” and press TAB. The New Salary field will fill in with “1000.” Fill in Raise as “200” and press TAB. The New Salary field will change to “1,200.”

Additional Information: You do not need to click Calc to cause the server to update the calculated field. PPR submits the values automatically when you navigate out of a changed field.

18. Close the browser. Stop the application.

What Did You Just Do?

You created a JSF page containing input fields and a submit button. The page used a backing bean that provided access to values with Expression Language expressions. Without PPR, you needed to submit the page so that the entire page, including the calculated value, would refresh. You then set up PPR by verifying that the input fields were assigned ID values, setting *AutoSubmit* on the input fields, and setting *PartialTriggers* on the read-only field. With PPR declared in this way, when you changed a value in an input field and navigated out of the field, the calculation was performed and the read-only field value changed accordingly, without having to submit the entire page.

This practice also demonstrated several commonly used ADF Faces atomic components (`af:inputText` and `af:outputText`) and layout containers (`af:panelFormLayout` and `af:panelLabelAndMessage`).

Caution

If you use PPR to set the *Rendered* property of a component to display or hide dynamically, you need to set the *Partial Triggers* property on a parent object such as an `af:panelGroupLayout` component that surrounds the component. Setting *Partial Triggers* on the component you are displaying or hiding will have no effect. This limitation applies only to the *Rendered* property.

How Can I Achieve the Perfect Layout?

Success in laying out a page that uses ADF Faces components requires at least a modest comfort level with the concept of using layout containers. While you are in the process of attaining that comfort level, it helps to allocate some design time to planning how you will use the layout containers in your application. Several tools will help you achieve the layout you require.

Facets

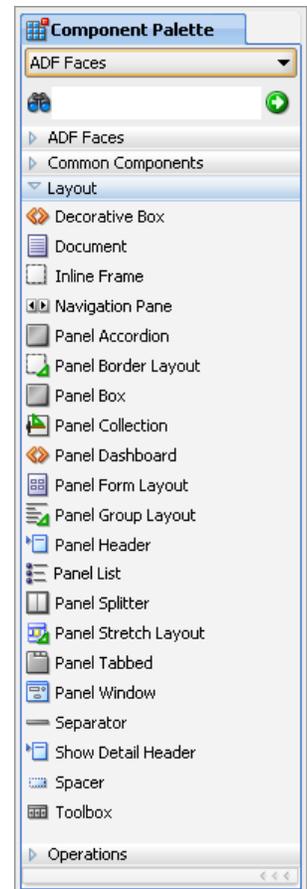
Layout containers offer *facets*, components into which you place other components. If the facet is responsible for a visual effect, at runtime, the facets maintain relative positioning of their contents. For example, the `af:panelFormLayout` layout container offers a footer facet that appears under all fields and is not bound by the alignment rules of the main area of the container. Facets is one tool you can use in achieving the perfect layout.

Note

Facets can contain only one component. However, the one component in a facet could be a layout container, which, in turn, can hold more than one component. JDeveloper leads by example in this case—if you drop a second component into a facet, instead of raising an error, JDeveloper will automatically add an `af:group` container around the two components. An `af:group` container component is just used to group related components into a single unit but it offers no visual properties of its own.

Knowledge of Layout Containers

The main tool you need to master for achieving the perfect layout is a knowledge of the layout containers. Fortunately, the list of layout containers is small at the moment as shown in the illustration on the right, from the Layout panel of the ADF Faces Component Palette page, where you can access most of the layout containers.



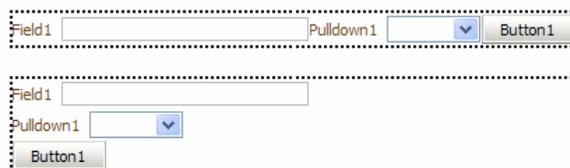
- af:panelFormLayout** As the PPR hands-on practice in this paper demonstrated, this container allows you to lay out fields in columns without requiring any other embedded containers. The fields are left-aligned, and their prompts are right-aligned. All fields are stacked on top of one another, but you can specify the number of columns and rows for a columnar arrangement. This provides a familiar appearance for data entry needs. The fields are rendered down one column and then down the next column. This ordering implements a *tab order* (the order in which the cursor navigates fields when the TAB key is pressed) of down and across. (The section “Nesting Layout Containers” later in this white paper discusses tab order.) The PPR hands-on practice shows screenshots of the effect of this layout container on its child components.
- af:panelHeaderLayout** This container provides a title to denote a grouping of elements on the page. It offers properties for *Size* to set the height of the heading text, *MessageType* to automatically add an icon (information or warning, for example) to the left of the heading text, *Text* for the heading text, and others. Nesting components within this container will stack them vertically on the page as shown next. The *InlineStyle* property has been set to “border-color:Black; border-style:dotted; border-width:thin;” to make the edges of the container visible.



Tip

Setting the background color as well as the border style of layout containers in this way is a useful debugging technique because it allows you to see the edges of each layout container.

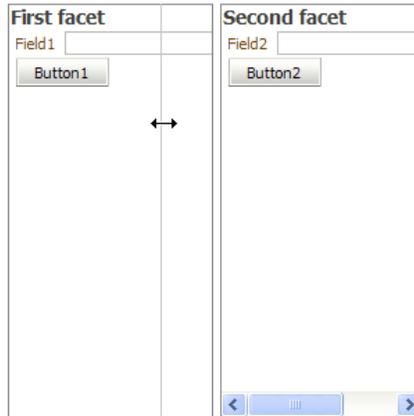
- af:panelGroupLayout** This container lays out components in a single row (horizontal layout) or column (vertical layout). The *Layout* property sets the orientation by which components in the container will be arranged: horizontal, vertical, or scroll (vertical with scrollbars). The following illustration shows a **af:panelGroupLayout** component with a *Layout* of “horizontal” and under it, another with the same components but with a *Layout* of “vertical.” As with the preceding illustration, *InlineStyle* defines a dotted line to show the edge of the containers.



Tip

af:panelGroupLayout works best when you explicitly assign a *Layout* property value other than “default” rather than relying on the default.

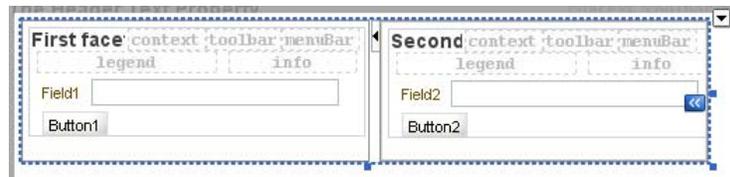
- af:panelSplitter** This container divides an area horizontally or vertically into two panels and optionally allows the user to move the dividing line between the panels to change the relative sizes of the panels. If the *Disabled* property is set to “true,” the user will not be able to move the splitter bar. You drop components into the first and second facets in this container to assign them to one panel or the other. For example, if you define a horizontal splitter, the user can move the splitter bar (divider) between panels to the left or right to make the first panel narrower or wider. The following illustration shows this effect as the splitter bar is in motion (the mouse cursor is the double-headed arrow):



The user can click the arrow icon on the splitter bar (not shown when the splitter bar is in motion, as it is in the preceding illustration) to collapse the first panel completely. Notice that the components in this display are truncated. To help you solve layout issues such as this, JDeveloper offers a pulldown menu in the top-right corner of the component after selecting it in the visual editor as shown here:



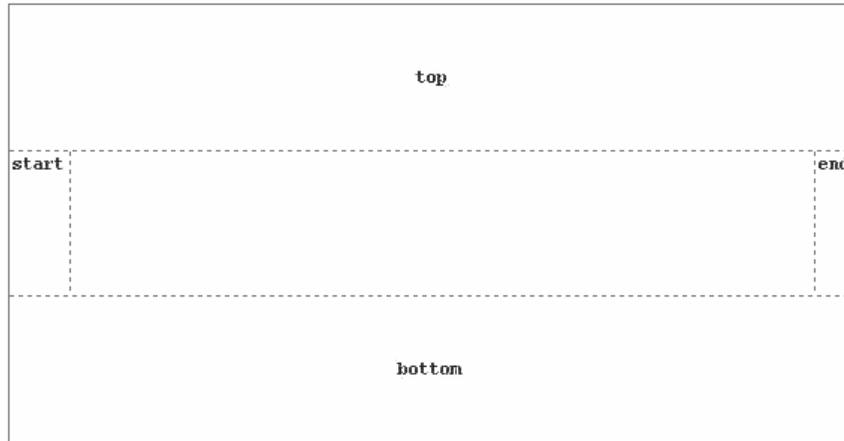
Selecting **Design This Container** opens an editor area that allows you to resize the component borders so that you could achieve a layout such as the following, where all components fit within the panels.



Note

This editor just sets *InlineStyle* and *SplitterPositions*, but it allows you to control the size visually instead of by experimenting with property values.

- af:panelBorderLayout** This layout container provides facets such as top, bottom, start, and end. Components dropped into one of these facets will maintain their relative positions at runtime. You can use this layout container for pages that require a logo in the header area (top facet) and copyright and informational text in the footer area (bottom facet). The facets are easiest to see in the visual editor as shown in the next screenshot.



- af:panelStretchLayout** This container also offers facets for bottom, start, top, and end. It also provides a center area, which stretches the contained component to fill the available width. You would use this container if another container cuts off the display of a layout container or atomic component. The following illustration shows this container with *StartWidth*, *EndWidth*, *TopWidth*, and *BottomWidth* properties set to “30.” Each facet contains an *af:panelGroupLayout* component with a background color of “Silver” and dotted border to better display the facet areas. The center area contains a button with no properties set except *Text* (the label). Notice how the center facet stretches the button to fill the available space.



The “Design This Container” editor described earlier allows you to set the size of the facets surrounding the center facet of the *af:panelStretchLayout* component as well its overall size.

Tip

The *af:panelGroupLayout* component is only as large as its child components. Therefore, if you have trouble aligning components within this layout container, it could be because the *af:panelGroupLayout* component is not wide enough. Try surrounding it with an *af:panelStretchLayout* component, which will expand the *af:panelGroupLayout* component and align its child components properly.

As a short demonstration of layout container properties and alignment capabilities, open the PPR application you created earlier in the white paper. Navigate to the Property Inspector for the *af:panelFormLayout* component and set *MaxColumns* (in the Common region) as “2” and *Rows* as “2.” Notice that the New Salary field moved into the second column because the tab (cursor navigation) order for an *af:panelFormLayout* layout container is across and down. The button retained its position at the bottom of the layout because it is placed in the footer facet.

Tip

If you try to copy and paste components in the Structure window but JDeveloper copies and pastes property values or other components instead, click a window such as the Log window to hide the Property Inspector. Then try the copy and paste operation in the Structure window again.

On the Salary field, set the *Columns* property (Appearance region) as “10” to set the width of the field. Notice that the New Salary field retains its horizontal position as shown here:

This alignment occurs because the New Salary field is in the second column, which is sized according to the widest field in the first column (Raise).

Other Layout Containers

Although most of the layout containers are available in the Layout panel of the Component Palette displayed earlier, you will find some additional layout containers in the Common Components panel as in the following examples:

- **af:form** This container renders a submit form (in HTML, `<form>`) used to pass content to the server for processing. It is technically not a layout container because it does not provide layout characteristics, but it is a container that holds other components.
- **af:menuBar** This component provides an area into which you place pulldown menu components. This component is called “Panel Menu Bar” in the Component Palette.
- **af:menu** This component displays a pulldown menu and can be placed into a popup or menubar.
- **af:popup** This container has no visual aspect; it displays whatever you place in it on top of the page.
- **af:dialog** This component displays a window containing a border, one or more buttons, and a content area. This component must be placed inside an `af:popup` component.
- **af:toolbar** This component displays an area into which you place command components such as buttons. This component is usually placed inside another layout container—`af:toolbox` (in the Layout panel), which contains toolbars and menubars.
- **af:panelLabelAndMessage** As demonstrated in the PPR practice, this layout container supplies a label to its contents. For example, the `af:outputText` component does not have a *Label* property. If you want to use that component to display read-only data values in a form, you can surround the component with `af:panelLabelAndMessage` and set the layout container’s *Label* property. Since you can place more than one component in a container, you could remove the labels on employee First Name and Last Name fields, surround those fields with `af:panelLabelAndMessage`, and fill in the container’s *Label* as “Employee Name” as shown here:

Employee Name

If the Employee Name structure is contained in an `af:panelFormLayout`, the container will treat it as a single field and apply the label and field layout alignments as shown next. The `af:panelFormLayout` is assigned a dotted border for visibility in this illustration.

Tip

Set Simple as “true” for `af:inputText` field components inside the `af:panelLabelAndMessage` container so that the field labels are not displayed. This setting also causes the fields to lay out horizontally as shown in the preceding illustration.

Visual Component Guide

The JDeveloper help system contains a visual representation of ADF Faces container (as well as all other) components. Enter “enhanced tag doc” in the help search field and click the link with the same name in the search results. Alternatively, navigate in the Contents tree to Javadoc and Tag Library Reference\JDeveloper Tag Library Reference\Oracle ADF Faces Tag Library Overview.

Nesting Layout Containers

As with other tag languages, such as HTML, you can embed layout containers within layout containers to take advantage of more than one type of arrangement. This is a useful and necessary tool for achieving the perfect layout. For example, the next illustration shows an `af:panelGroupLayout` container (defined with *Layout* as “vertical”) that surrounds three more `af:panelGroupLayout` components (defined with *Layout* as “horizontal”), each of which contains two `af:inputText` components. The *Columns* properties of the fields have been set differently to show the somewhat ragged alignment that will result.

Tip

Use the `af:spacer` component to add a fixed horizontal or vertical space between components. Enter values in the `af:spacer` properties “Width” and “Height” to designate a pixel width or height for the spacer, for example, “10px.”

The tab order for an `af:panelGroupLayout` with *Layout* set to “horizontal” is across. The tab order for an `af:panelGroupLayout` set to “vertical” is down. Therefore, the preceding arrangement implements a tab order of across and down—within a inner horizontal `af:panelGroupLayout` component and between inner horizontal `af:panelGroupLayout` components. This type of layout provides an alternative to the `af:panelFormLayout` tab order of down and across. Techniques for implementing a tab order of across and down are explored in the following hands-on practice.

Hands-on Practice: Implement an Across and Down Tab Order

If you wanted to emulate the alignment characteristics of a `af:panelFormLayout` layout container in addition to implementing an across-and-down tab order, you could use these two alternative techniques:

I. Use spacers to align the fields**II. Use container properties to align the fields**

This hands-on practice demonstrates both of these alternative techniques and mentions some of the benefits and drawbacks of each.

I. Use Spacers to Align the Fields

This technique consists of adding `af:spacer` components before and between fields to achieve the following layout:



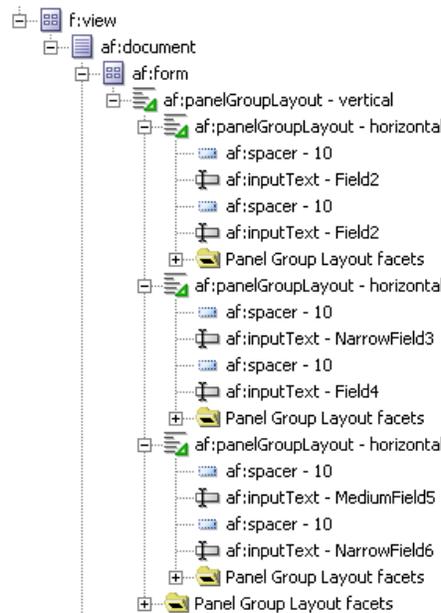
This arrangement looks similar to the arrangement of components within an `af:panelFormLayout` container. It has a more complicated setup (finding the widths required for each spacer takes experimentation) but allows you to provide a tab order that some users prefer.

1. In the ADFFacesTest application you used for the earlier hands-on practice, create a JSF page called tabOrder1.jspx. Specify that JDeveloper will not create a backing bean file.
2. Drop a Panel Group Layout component into the `af:form`. Set *Layout* to “vertical.” Drop three more Panel Group Layout components on top of the first `af:panelGroupLayout` and set *Layout* for the three components to “horizontal.” This will result in three rows inside the outer layout container.
3. In the first row, drag a Spacer component on top of the `af:panelGroupLayout` and set its *Width* property to “22”; then add an Input Text component and set *Columns* to “35”; add a Spacer and set its *Width* to “25”; and add a final Input Text and set *Columns* to “10.”

Repeat this process (using copy and paste if you want to) for the other two rows using the settings in the following table.

Row	Field (<i>Columns</i>) or Spacer (<i>Width</i>)
2	Spacer (11); Input Text (10); Spacer (150); Input Text (10)
3	Spacer (10); Input Text (20); Spacer (65); Input Text (5)

Additional Information: The Structure window should appear as follows (although the Structure window labels for your fields may be different at this point). Notice that the `af:spacer` nodes show the *Height* property (all defaulting to 10), not the *Width* property that you set. The *Columns* property represents character columns, not pixels.



4. Set the field labels as shown in the preceding illustration. The layout editor will not show the final runtime alignment although the **Preview in Browser** right-click menu option will.
5. Run the file to test the field alignments. Proper alignment will depend upon the skin you are using (discussed in the section “Working with Skins” later on). If needed, so adjust the properties so the field alignment matches the earlier illustration. Resize the window width to test variations on the window size.

Additional Information: Since the field labels are single words, the alignment will be maintained. If you change the labels to multiple words, they would wrap if the window width became too small; this would disrupt the field alignment. Since the field labels are used to maintain field alignment, if the field labels need to be changed dynamically based on the user’s language, the work required for this technique becomes nearly unmanageable.

6. Close the browser.

What Did You Just Do?

You set up nested layout containers with specific alignments to implement an across and down tab order. The problem with this layout arrangement is that it requires experimentation to find the correct size for the spacers. Another problem mentioned before is the dependency on the width of field labels, which may change based on the user's language and which are subject to wrapping if the window is narrowed sufficiently.

Caution

This alignment technique may not produce consistent results on all versions of all browsers. It is always a good idea to try your application with different browsers to verify the visual effects.

II. Use Container Properties to Align the Fields

An alternative technique for implementing an across and down tab order, but still maintaining the alignment of fields between rows, is to use multiple `af:panelFormLayout` components, each of which displays a single row. If you set each row's layout container with the same space allocated to the field width and label width, you can achieve the vertical alignment of fields, even if the fields and field labels are different widths. The following steps use this technique to create the layout shown in Figure 1.

Multiple Panel Form Layouts

Narrow Field 1

Wide Field 2 with a wrapping label

Medium Field 3

Medium Field 1

Narrow Field 2

Wide Field 3

Single Panel Form Layout

Narrow Field 1

Medium Field 1

Wide Field 2 with a wrapping label

Narrow Field 2

Medium Field 3

Wide Field 3

Figure 1. `af:panelFormLayout` component tab order demonstration

Caution

This is not a complete functional equivalent to the single `af:panelFormLayout` solution. We explain the drawbacks of this technique after the hands-on practice.

1. Create a JSF page called `tabOrder2.jspx` (no managed bean). You will be adding components to create the structure shown in the Structure window snippet in Figure 2.
2. Drop a Panel Header on the page and set *Text* to "Multiple Panel Form Layouts."
3. Drop a Panel Group Layout into `af:panelHeader` and set *Layout* as "vertical." This component will supply the mechanism to stack rows of fields.
4. Drop a Panel Form Layout component onto the `af:panelGroupLayout` component and set *MaxColumns* as "2" and *Rows* as "1."
5. Select the `af:panelHeader` and copy it. Paste it onto the `af:form` component to create another region. Change *Text* of the copy to "Single Panel Form Layout."
6. In the Multiple Panel Form Layouts region, drop two Input Text components into the `af:panelFormLayout` component.

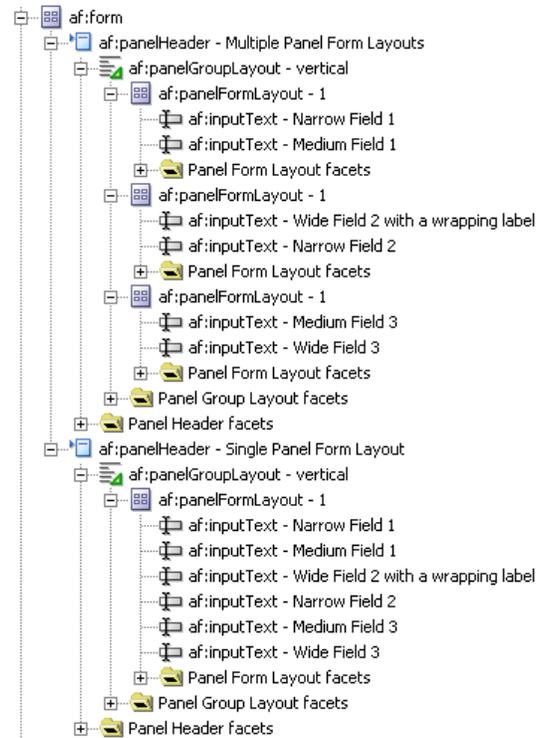


Figure 2. Structure window view of the tab order demonstration

7. Copy the `af:panelFormLayout` component and paste on top of the `af:panelGroupLayout` component to create the second row of fields.
8. Repeat the preceding step to create the third row of fields.
9. Group the three `af:panelFormLayout` components and set the following properties:
 - FieldWidth* as “300px”
 - LabelWidth* as “100px”
 - Inline Style* as “width:800.0px;”

Additional Information: The *FieldWidth* property designates how much space is available for each input item in the layout container. The *LabelWidth* property sets a fixed space for each input item’s prompt. The *Inline Style* width sets the available space for the entire layout container. The amount of space allocated to each column is half of that (400 pixels in this example) because we are specifying two columns.

10. Change the `af:inputText` *Label* properties to match those shown in Figure 1.
11. Group the two components with “Narrow” labels (using CTRL-click) and set *Columns* as “10.”
12. Group the two components with “Medium” labels and set *Columns* as “25.”
13. Group the two components with “Wide” labels and set *Columns* as “50.”
14. Under the the Single Panel Form Layout `af:panelGroupLayout`, set the `af:panelFormLayout` *Rows* as “3.”

Additional Information: *MaxColumns* is the important setting, but it is good practice to set *Rows* as well. Rows will be formed by the layout container placing the *MaxColumns* number of items on each line. If the number of items exceeds *MaxColumns* times *Rows*, the number of rows will be extended.

Note

Be sure the *FieldWidth*, *LabelWidth* and *InlineStyle* properties are blank for the Single Panel Form Layout `af:panelFormLayout` component.

15. Group all six fields from the three `af:panelFormLayout` components in the multiple panel region above and copy them, then paste them onto the `af:panelFormLayout` in the Single Panel Form Layout region. Compare your Structure window nodes with the illustration shown in Step 1.
16. Click Save All. Run the page.
17. Click in the first field and tab through the six fields in the top region to verify that the tab order is across and down.
18. Continue tabbing through the fields in the bottom region to verify that the tab order is down and across.
19. Resize the window width. Notice that the fields in the top region maintain their relative positions but they do not move based on the window width. The fields in the bottom region react to the window resizing and relocate so that more of the content is visible.
20. Close the browser. Stop the application.

What Did You Just Do?

You created a tab across-and-down region and a tab down-and-across region each with six fields. The former layout technique uses multiple `af:panelGroupLayout` and `af:panelFormLayout` components to emulate the alignment effect of a single `af:panelFormLayout` component. The resulting layout is not as fragile as the technique demonstrated in the preceding practice and it does not require as much experimentation. However, it has some limitations: mainly that the container widths need to be set explicitly. This takes a small amount of calculation and experimentation. Also, if the user increases the browser font size or uses a different skin, the width of one of the fields may exceed the width allotted and one or more rows may lose their alignment with the other rows. A summary of the differences in features between the single `af:panelFormLayout` component technique and the multiple `af:panelFormLayout` technique appears in Table 1.

Feature	Single	Multiple
Tab order	Down and across	Across and down
Fields left-align and labels right-align	Yes	Yes
Width of the browser window determines width of the container	Yes	No
Width of a column within the container is based on the width of the widest field in that column	Yes	No
Calculations and experimentation are needed to assign widths	No	Yes
Technique is immune to the user increasing the browser font size	Yes	No

Table 1. Comparison of Features for Single Panel Form Layout and Multiple Panel Form Layout Containers

Should you decide to use this technique, you will need to experiment with setting the *FieldWidth* property (in pixels) to accommodate the widest field. Fields are sized in character columns, which does not translate to pixels, and this is why you need to experiment a bit. Remember that you need to multiply the sum of the largest field width plus a reasonable label width by the number of fields per line to reach the *InlineStyle* width for the `af:panelFormLayout` component. In other words, the formula to use (after finding the optimal field width and label width) follows (all widths use the unit pixels):

$$(\text{FieldWidth} + \text{LabelWidth}) = (\text{Inline Style:Width} / \text{Number of Fields per Line})$$

Quick Start Layouts

One way to shortcut work with layout containers is to create the page using Quick Start Layouts. *Quick Start Layouts* are pre-built sets of layout containers that are designed for specific functionality. You access this feature from the Create JSF Page dialog (started from the New Gallery's JSF Page item). Selecting Click Start Layout in the dialog and then clicking Browse will open a dialog such as that shown in Figure 3.

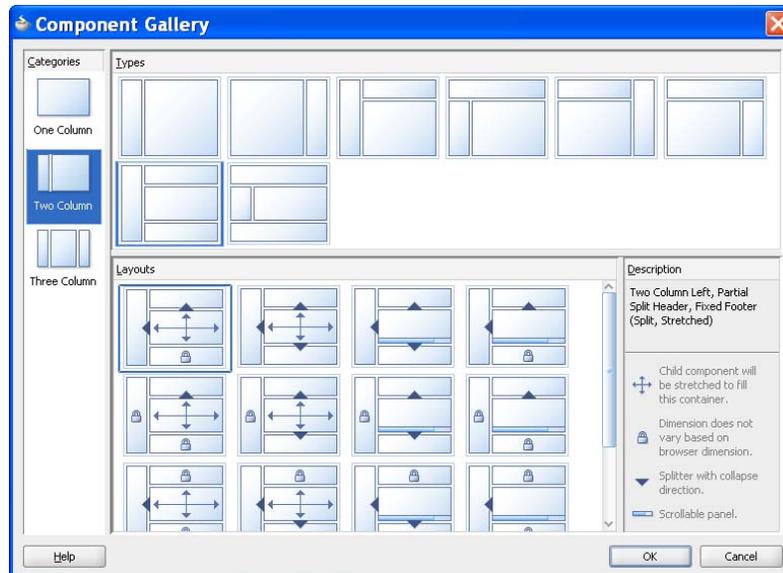
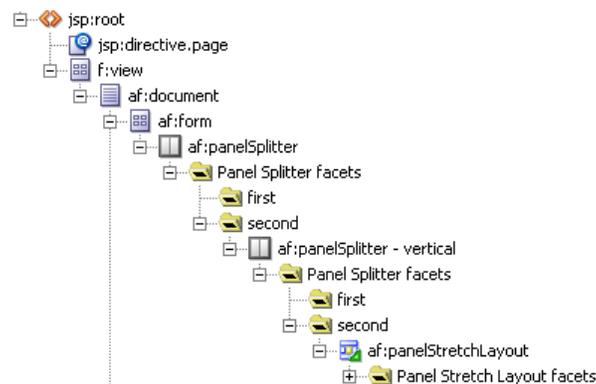


Figure 3. Quick Start Layout Component Gallery window

Selecting a category in the Categories panel and a type in the Types panel will display a number of options in the Layouts panel. Clicking a layout will display a short description about its layout. After selecting a layout, you click OK. JDeveloper will create the JSF page with a set of layout containers based on the layout you selected. For example, selecting “Two Column Left, Partial Split Header, Fixed Footer (Split, Stretched)” will preload the following structure onto the new JSF page:



You then drop components into the pre-built layout containers.

Becoming familiar with this feature can save you some time in laying out the proper layout containers. If you study the resulting layout from a Quick Start Layout selection, you are likely to learn a bit about how to properly nest layout containers.

What Types of Atomic Components Are Available?

In addition to the layout container components, ADF Faces supplies a large number of atomic components that render as user interface elements such as input fields, selection lists, date and color pickers, menus, trees, tables, shuttle controls, and buttons. Figure 4 shows the Component Palette window with the Common Components panel (split into four columns to

save space, although the Component Palette is actually a single column). Most of the components in this list are atomic components, although, as mentioned, some are layout containers.

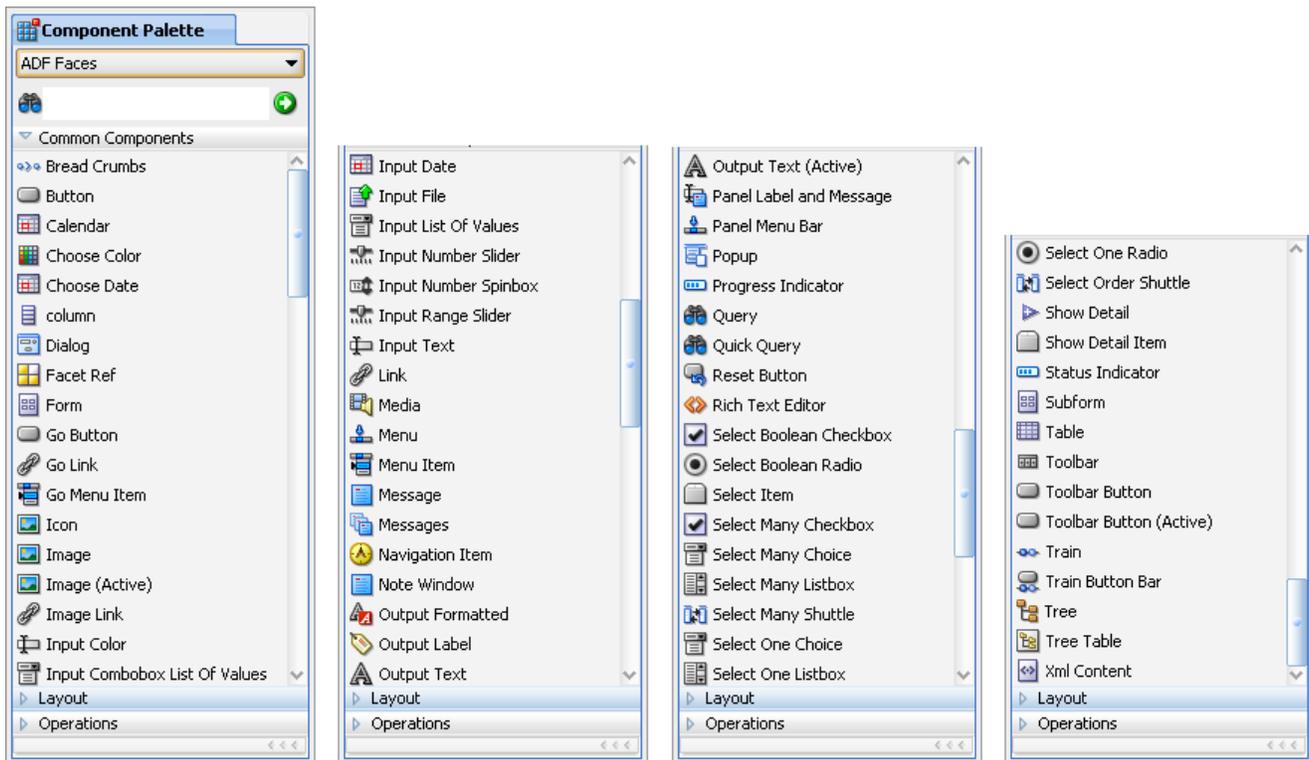


Figure 4. ADF Faces Component Palette page, Common Components panel

Describing all of these components would not only take more pages than are allocated for this white paper, but it would also repeat a lot of good information available elsewhere, for example, the tag library visual overview mentioned in the earlier tip. However, it is useful to get a taste for the some of these components. The visual representation of the components described is shown in Figure 5.

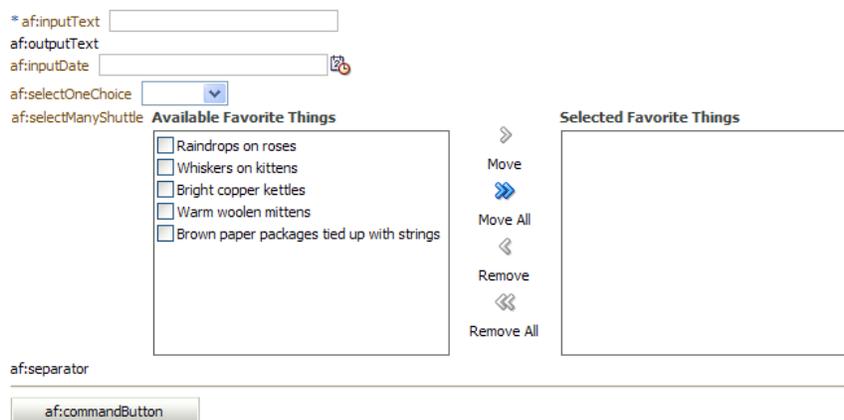
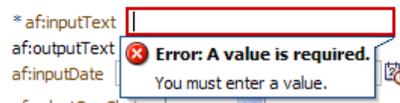
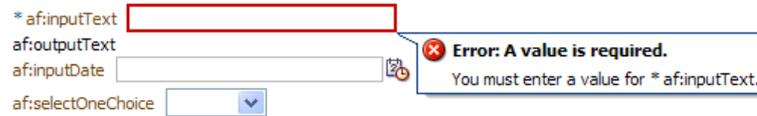


Figure 5. Some ADF Faces atomic components at runtime

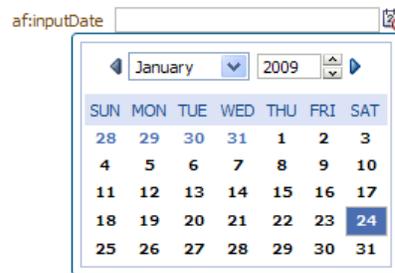
- af:inputText** This component is a standard text item that includes a label (prompt) property. Another of its many properties is *Required*; if this property is set to “true,” the framework will validate that a value is entered when the page is submitted. As shown in Figure 5, it will also display an asterisk (“*”) before the label as a hint to the user that this field value is mandatory. If the user submits the page without a value, a message such as the following appears:



You can customize the message using the *RequiredMessageDetail* property. For example, suppose *RequiredMessageDetail* is set to “You must enter a value for {0}.” The following will appear if the page is submitted. (The item label fills in for the “{0}” placeholder.)



- **af:outputText** This component displays a read-only field and is handy for text that the user will never be able to edit. The component does not offer a *Label* property, but you can embed it inside an *af:panelLabelAndMessage* layout container and set the *Label* property of the layout container if needed. If you will allow users to enter data into the field conditionally (for example, based on the user’s role or other values), use an *af:inputText* component instead and set that component’s *Disabled* property conditionally.
- **af:inputDate** This component displays a text item with a graphical LOV button as shown in the next illustration. When the user clicks this button, a calendar window will appear as in the illustration on the right. The user can select a date from this calendar window, and the selected date will be returned into the associated text item.



- **af:selectOneChoice** This component displays a standard pulldown item. After dropping an *af:selectOneChoice* component on the page, you will be prompted to define the source of the data in the pulldown. In some cases, your application might base the list on a database query, but you can also create a static list that will be embedded into the component tag as shown in this example:


```
<af:selectOneChoice label="af:selectOneChoice" id="soc1">
  <af:selectItem label="Choice1" value="1" id="si2"/>
  <af:selectItem label="Choice2" value="2" id="si1"/>
</af:selectOneChoice>
```
- **af:selectManyShuttle** This component displays a shuttle control—two text areas that allow the user to select more than one value for a single field. All headings, graphics buttons, and text areas are built into this component. Child components of this component define the values that appear in the available (left-hand) list.
- **af:separator** This component renders a horizontal line to divide a page horizontally. (The *af:separator* label in Figure 5 is an *af:outputText* item, not part of the *af:separator*.)
- **af:commandButton** This component displays an action button. When the user clicks this item, an event is triggered defined in the *Action* or *ActionListener* properties. For example, you define a navigation case from page1 to page2 in the faces-config.xml file and name the outcome “openPage2.” You can then set the *Action* property for a button on page1 as “openPage2” so that when the user clicks the button, page2 will open.

Other Atomic Components

The most commonly used atomic components appear on the Common Components panel of the ADF Faces Component Palette (shown in Figure 4). The Component Palette also provides an Operations panel (shown on the right) that lists more atomic components, most of which are not visual in nature but are used in conjunction with other atomic components.

The online help system's visual tag library page contains links that navigate to the tag documentation for each component. Components are organized into three categories in this help page: converters, validators, and miscellaneous. As mentioned, you can access the tag library help page using the Contents tree Designing and Developing Applications\Developing Oracle ADF Applications\Oracle ADF Faces Tag Library Overview node. This page shows and briefly describes the ADF Faces tags (over 130 at last count).

Converters

These components nest under data value components such as `af:inputText`. They transform the inherently string data type for HTML text fields to a number (`af:convertNumber`), date (`af:convertDateTime`), or color (`af:convertColor`, for red-green-blue value combinations). You use the *Pattern* property of the converter tag to specify the format mask.

Hands-on Practice: Test Converters

As an example, try these steps in JDeveloper in the `ADFFacesTest` application:

1. Create a JSF page called "operationsDemo.jspx." In the Page Implementations area, specify that no managed bean will be created. Drop two Input Text items onto the page. Drop a Button under the second field (so that you can submit the page in a later step). Set the `af:commandButton Text` property as "Test." Run the page.
2. Enter a mixture of numbers and characters in the first field and tab to the second field. No validation occurs. Leave the browser open.
3. In JDeveloper, from the Operations panel, drop a Convert Number component on top of the first field in the Structure window. Repeat the drop operation for the second field.
4. With the second field's `af:convertNumber` node selected in the Structure window, set *Pattern* as "`###,###.##`" without quotes (the Java-style format mask for this converter). This format will be validated on the server when the value is submitted.

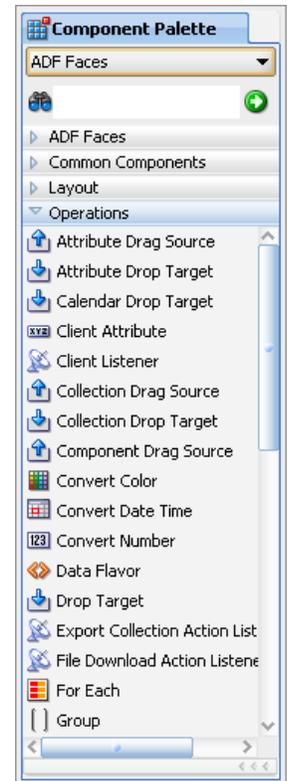
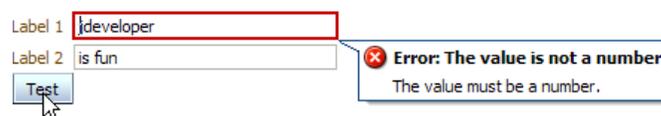
Additional Information: The Javadoc for the `java.text.DecimalFormat` class contains details about the available number format mask characters.

5. Click Save All. Refresh the browser.
6. Enter alphabetic characters in the first field and press TAB. The field border will change to red, indicating a value error, and the hint will appear for the second field to indicate the proper format as shown next.



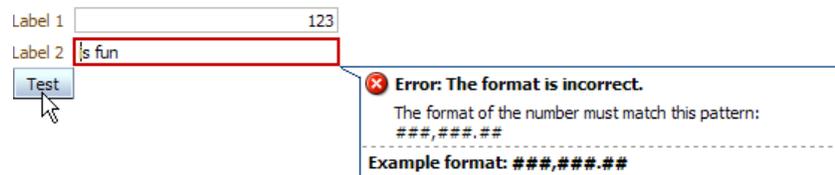
Additional Information: You can change this and other messages by setting properties on the converters and validators (explained in the following sections).

7. Enter alphabetic characters in the second field and click TAB. The cursor will move to the button, but the second field's border will not indicate a value error. Click Test. The cursor will move to the first field and the following error message will appear:



Additional Information: The validation for the second field occurs on the server side (after you submit the page) because the *Pattern* property is not supported on the client side.

8. Change the value in the first field to a number and click Test again. The following message will appear:



Note

Notice that the number value in the first field is right-aligned, whereas when you typed an alphabetic string, the value was left-aligned. This type of alignment is usually preferred so that stacked number fields can be more easily compared.

9. Change the second field's value to a number and click Test. The error state will be cleared and no messages will appear.
10. Close the browser.

Tip

Set the *IntegerOnly* property of `af:convertNumber` to "true" if the value must be an integer. This property is validated on the client side.

Validators

Validator components check component values for different aspects: range (`af:validateDateTimeRange`, `af:validateDoubleRange`, and `af:validateLongRange`); length (`af:validateByteLength` and `af:validateLength`); date characteristics (`af:validateDateRestriction`); and regular expression (`af:validateRegExp`).

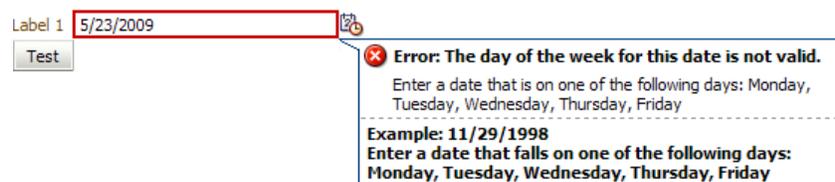
Hands-on Practice: Test Validators

Try these steps on the page you just used to test the number converter:

1. In the `operationsDemo.jspx` page, drop an Input Date component above the button. Drop a Validate Date Restriction (Operations) component on top of that component.
2. Set `InvalidDaysOfWeek` as "sat sun" and press ENTER. Click Save All.
3. In the Log window's Running tab, one of the messages in the last section will start with "Target URL." Click the HTTP link next to that label. The browser will reopen and the page will load. The new date field should be displayed.

Additional Information: Since you changed the page and saved the changes, the timestamp of the file was updated. The server determined that the servlet already created for that page was older than the source JSP code and therefore recompiled it before running it. If you do not see the date field, stop the application and try this step again.

4. Move the cursor into the date field. A hint will appear indicating that a date other than Saturday and Sunday should be entered. Type in a Saturday or Sunday date and press TAB. A client-side error message such as the following should appear:



Additional Information: Clear the value, and then click the calendar icon to the right of the field. The calendar (date picker) popup will appear. Selecting a Saturday or Sunday date from the calendar will cause the error message just displayed to appear without your having to press TAB.

5. Close the browser.

Tip

Another operations component, `af:validateRegExp`, uses the power of regular expressions to validate the format of the value. It is useful for enforcing the format of string values, for example, Social Security Number, telephone number, or postal codes such as the six-digit codes used in Canada (for example, "K1S 4W7"). The `af:validateLength` component can help validate a postal code such as the base five-digit zip code in the United States that can be left-padded with zeros.

Miscellaneous

This category of the Operations panel in the ADF Faces Component Palette page includes components with varied functionality, for example:

- **af:forEach** This component supplies iterative logic within the JSF page. It replaces the JSTL equivalent tag for ADF Faces work.
- **af:pageTemplateDef** This component serves as the root component in a page template. The Create JSF Page Template dialog creates a page containing this tag, but you can create templates manually if needed by dragging this tag from the Component Palette.
- **af:showPopupBehavior** This component displays a popup (named in the *PopupId* property).
- **Drag and drop components** When declaring drag-and-drop behavior, you use Operations tags such as `af:attributeDragSource`, `af:attributeDropTarget`, `af:clientAttribute`, `af:collectionDragSource`, `af:collectionDropTarget`, `af:componentDragSource`, and `af:dropTarget`. The next section steps through setting up drag-and-drop behavior in an ADF Faces application.

Hands-on Practice: Set Up Drag and Drop Behavior

The Operations tab includes tags you can use to set up drag-and-drop actions in your application. This allows users to assign values to items based on values of other items. The following brief set of steps demonstrates how to use three drag-and-drop components.

1. Create a JSF page called `dragDropDemo.jspx` and for Page Implementation, specify that a managed bean will be created.
2. Drop in a Panel Group Layout. Into the `af:panelGroupLayout` component, drop the following components in this order, setting their properties as indicated:
 - Input Text (*Label* as "Drag Source" and *Value* as "2000")
 - Button (*Text* as "Submit")
 - Separator
 - Spacer (*Height* as "50")
 - Input Text (*Label* as "Drop Target")

Additional Information: The Submit button is required because the value you type into the Drag Source field must be submitted to the server so that it can be used to set the value of the drop target. The initial value of the Drag Source item ("2000") can be used right after the form starts because it is already set as the component value on the server side.

Note

The Common and the Appearance sections of the Property Inspector display the same *Text* property.

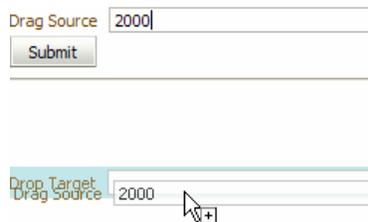
- Run the page. Try dragging the Drag Source field. It will not move because you have not added drag-and-drop operations yet. Leave the browser open.
- Drag an Attribute Drag Source from the Operations panel onto the Drag Source field. Set *Attribute* as value. Click OK.

Additional Information: This `af:attributeDragSource` component defines the Input Item as the origin of a drag-and-drop operation.

- Drag an Attribute Drop Target from the Operations panel onto the Drop Target field. Set *Attribute* as “value.” This attribute specifies which property of the Drop Target field will be filled in by the drag-and-drop operation. Click OK.

Additional Information: The `af:attributeDropTarget` component defines the destination for a drag-and-drop operation.

- Click Save All. Refresh the browser. Grab the Drag Source field and drop it onto the Drop Target field as shown here:



Additional Information: The value will be copied to the Drop Target field.

- Modify the Drag Source value and drop it onto the Drop Target again. The value will not change. Click Submit and try the drag-and-drop operation again. This time the value will change because the new value has been submitted to the server and has updated the component’s value.
- Close the browser.

Data Visualization Tools (DVT)

Another class of atomic components raises the bar on high interactivity—the *Data Visualization Tools (DVT)* tag library. DVT consists of components that you can use to display data visually such as the following:

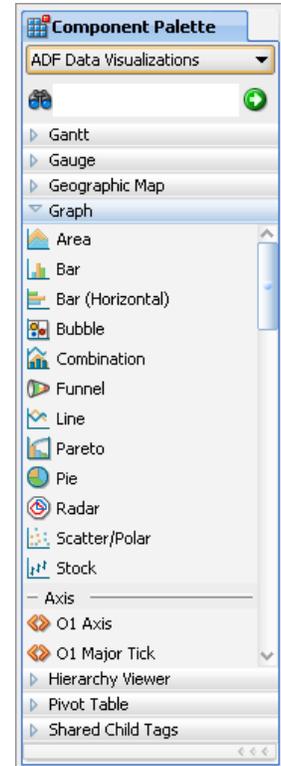
- Gantt chart** This component is used to track project work.
- Gauge** This component displays minimum and maximum values for a data range and places a data point within that range.
- Geographic map** This component displays a map and allows you to place data on the map. The user can use zoom and pan operations to access different views of the map.
- Graph** This component allows you to represent the relations between sets of data in nearly any style, for example, area, bar, bubble, funnel, line, pareto (single-series bar graph), pie, radar, scatter/polar, and stock.
- Hierarchy viewer** This DVT component displays parent-child relationships between data elements and allows the user to pan and zoom in the display. The data elements can support drill-down operations so that double clicking a node can open a window or dialog with more information.
- Pivot table** This element displays an interactive spreadsheet that allows the user to drag and drop data to create summaries or sorts dynamically. This type of component is useful for multi-dimensional data or data used primarily for analysis.

The illustration on the right shows the ADF Data Visualizations page of the Component Palette with the Graph panel partially expanded to show the major graph components. Each panel contains one or more major component and a number of child components that you place into the major component.

You can try out the following hands-on practice to get a taste of how to work with the DVTs.

Hands-on Practice: Build a Bar Graph

In this practice, you will create a simple bar chart that shows the sum and average salaries in each department. You will modify several default properties to see how to work with the graph DVT component. In this practice, notice that when you drop a data collection from the Data Control panel, you will see options for several DVT style components as shown here:



1. Create an application, called “DVTGraph” using the Fusion Web Application (ADF) template.
2. On the Model project node select **New** from the right-click menu. Double click the View Object item in the ADF Business Components category.
3. The Initialize Business Components Project dialog will appear. Create a connection or use an existing connection to the HR schema. Click OK.

Caution

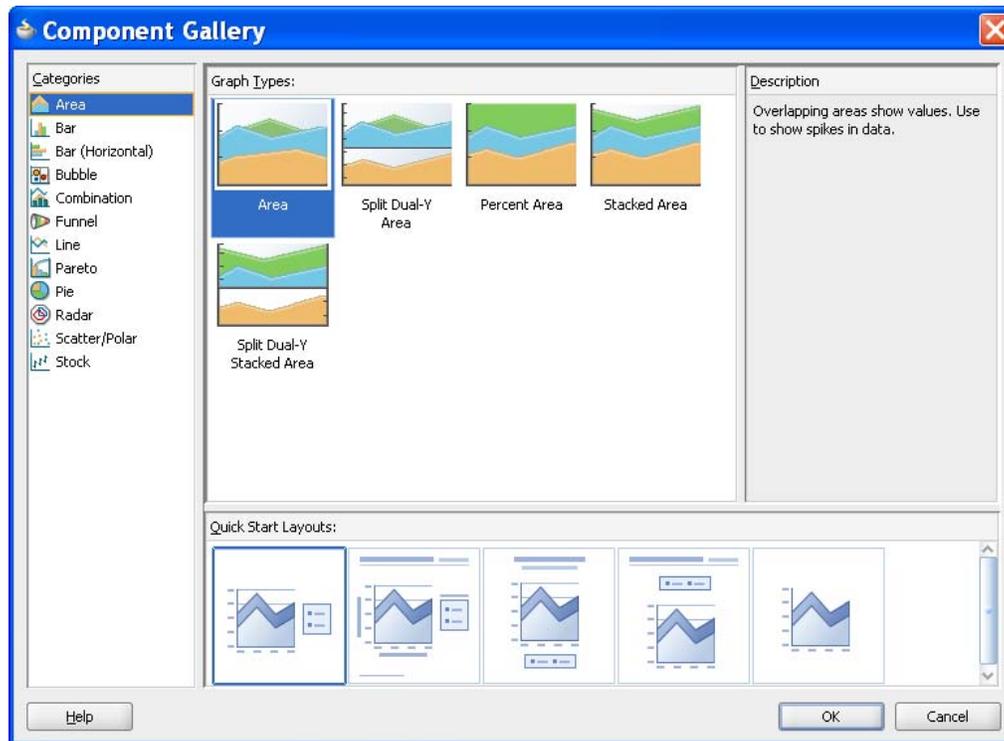
If you receive an error that the connection cannot be created, click Save All, restart JDeveloper, and continue from this point.

4. In the Create View Object Wizard, enter *Name* as “DepartmentSalary.” Select “Read-only access through SQL query” and click Next.
5. Fill *Query Statement* as follows:

```
SELECT SUM(emp.salary) AS sum_salary,
       AVG(emp.salary) AS avg_salary,
       dept.department_id, dept.department_name
FROM   departments dept, employees emp
WHERE  dept.department_id = emp.department_id
GROUP BY dept.department_id, dept.department_name
```
6. Fill *Order By* as “department_name.” Click Next, Next, and Finish.
7. On the demo.model node of the Model project select New Application Module. Fill in the wizard fields to create an application module called “HRService” and add an instance of DepartmentSalary to the data model.

Additional Information: On the Step 2 page, move DepartmentSalary to the Data Model area and click Finish. Click Save All.

8. In the ViewController project node, create a JSF page called “deptSalaryGraph.jspx.” For the Page Implementation section, specify that no managed bean will be created.
9. Find the Data Controls panel in the Application Navigator window. If the Application Navigator window does not appear, select it in the View menu. Drop DepartmentSalary1 from the Data Controls panel onto the page and select **Graphs**. (Select **View | Refresh** or collapse and expand the Data Controls panel if the Data Controls panel is empty.) The following dialog will appear:



10. Select “Bar” in the *Categories* list and select “Bar” (the default) in *Graph Types*. Select the first selection in the *Quick Start Layouts* area (the default). Click OK.
11. Click the Add Attribute (green plus “+”) button next to the *Bars* field and select “SumSalary.” Click the same button and select “AvgSalary” to add that attribute to the *Bars* list. These two query columns will appear as bars in the graph.
12. In the Attribute Labels area, change the SumSalary label to “Total of All Salaries.” Change the AvgSalary attribute label to “Average Salary.”
13. Click Add Attribute next to the X-Axis field and select “DepartmentName.” This declares that the department name will show as labels on the horizontal axis. The dialog should appear as in Figure 6.
14. Click the Preview tab to see how the graph will appear. Click OK to dismiss the dialog.
15. Click Save All. Run the page. Move the mouse over a bar to view a popup with the data details.
16. Leave the browser open and return to JDeveloper.

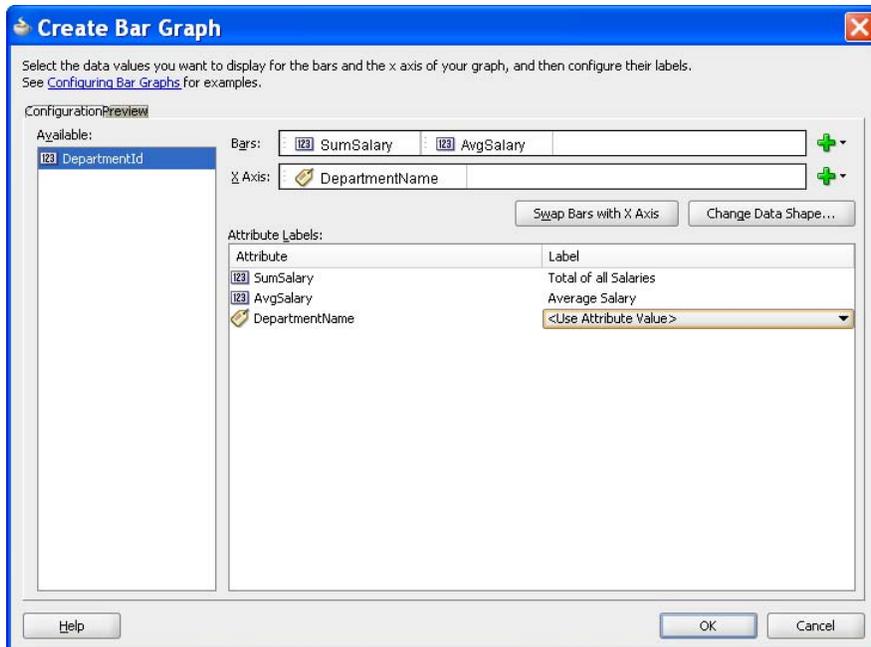
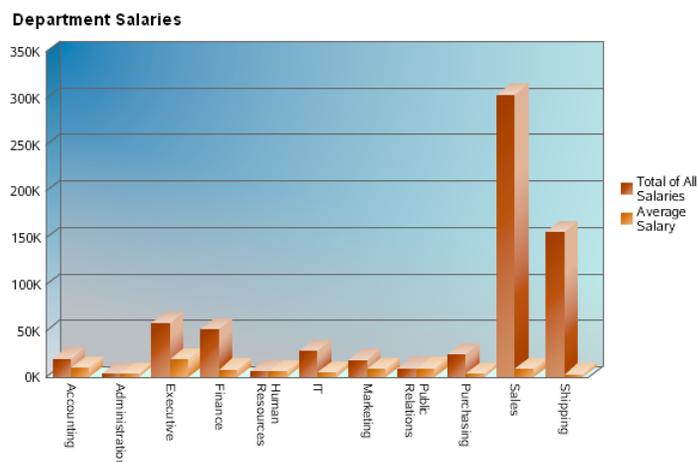


Figure 6. Selecting bars and x-axis in the Create Bar Graph dialog

17. In the Design tab of the editor, click the graph component to select it. In the Property Inspector, Graph Data section, notice that the *Value* expression refers to a graphModel binding (“#{bindings.DepartmentSalary1.graphModel}”). Navigate to the Page Binding file (in the editor window, select **Go to Page Definition** from the right-click menu in the editor) and notice in the Source tab that the binding DepartmentSalary1 is defined as a graph binding (<graph />).
18. In the deptSalaryGraph.jspx editor, select the graph component again. In the Appearance section of the Property Inspector, change the following properties and notice the changes in the visual editor along the way:
 - Style* as “Southwest”
 - Title* as “Department Salaries”
 - ImageHeight* as “400”
 - ImageWidth* as “600”
 - 3D Effect* to “true”
19. Click Save All. Refresh the browser to view the changes (press the SHIFT key while refreshing to force a full refresh). The graph should now appear as follows:



20. Close the browser. Stop the application.

What Did You Just Do?

You sampled the development experience when working with a DVT component. You created a read-only view object as a SELECT statement to retrieve the total salaries for each department. You also created an application module to expose the view object to the View layer code. Then, you defined a bar graph based on the view object and changed some properties of that graph.

What Could You Do Next?

To continue the experiment with the graph component, you can set up a master-detail drill-down feature using the following abbreviated steps. The effect will be that clicking a department bar in the graph will display a table of employee records for that department.

1. Create an entity object definition, Employees, based on the EMPLOYEES table and specify a default view object, EmployeeDetails. This will serve as the data source for the detail table.
2. Create a view link definition from DepartmentSalary (source) to EmployeeDetails (destination) based on DepartmentId. Open the HRService application module and add a detail instance of EmployeeDetails naming it "EmployeesInDepartment" as shown here.
3. In the Application Navigator, refresh the Data Controls panel (**View | Refresh** after selecting a node). Select the deptGraphSalary.jspx document tab and drop the EmployeesInDepartment nested data collection on the af:form node of the Structure window. Select the **Tables | ADF Read-only Table** option. In the Edit Table Columns dialog, delete all but the EmployeeId, FirstName, LastName, Salary, and DepartmentId columns. Click OK.



Additional Information: The DepartmentId column is included only for testing so that you can see that the same DepartmentId is retrieved for the click on a bar. You now need to set two properties to cause the table to refresh when a department bar is clicked.

4. Select the af:table node in the Structure window, and set *PartialTriggers* as "::barGraph1" (the bar graph).

Note

The *fully qualified name* of a component refers to its relative path in the component hierarchy with each node delimited by a colon (":"). In this example, you need to refer to the ID of the top-level component "barGraph1" in a *PartialTriggers* property, so refer to it as "::barGraph1." References to sibling components (components on the same level) do not need the path prefix.

5. On the graph component, set *ClickListener* (Behavior section) to "#{bindings.DepartmentSalary1.graphModel.processClick}." (You may need to type in this value if the Expression Builder is not available for this property.) The method `processClick()` is automatically available to graphs.
6. Click Save All. Run the page. The employee table will appear under the graph. Click a bar in the graph and the employee table will refresh with the related employee records.

Additional Information: The table appears truncated on the right side. The reason is that the table component does not automatically fill the available space. The solution is to surround the table with an `af:panelStretchLayout`, whose purpose is to cause its child components to fill the available space.

7. Leave the browser open and return to JDeveloper. In the Structure window, drop a Panel Stretch Layout component onto the `af:form` component so that it appears under the table. Then drag the existing `af:table` into that container's center facet. Click Save All and refresh the page. The table should better fill the screen.
8. Close the browser and stop the application.

How Can I Change the Default ADF Faces Appearance?

Although developers are often impressed (regardless of whether they acknowledge it) with ADF Faces' rich features and extensive set of components, one of their first reactions (after being impressed) is to question how to change the appearance of the components. ADF Faces components are bound to a default set of Cascading Style Sheets (CSS) files and a resource bundle that defines an extensive set of selectors. These style sheets provide a consistent, well-designed, common look and feel to all components by default. This means that every `af:inputText` component that you use in an application will share the same visual characteristics.

The problem is that the default may not agree with the standards set up for your applications. The easiest way to solve this inconsistency is by changing your standards to match the ADF Faces defaults. This is the “WYSIWYG-IWYGIWYW” principle: What you see is what you get if what you get is what you want. That is, the default ADF Faces look-and-feel standards will serve you perfectly if your standards match the ADF Faces standards.

Naturally, that principle is easy to state, and although it seems a bit flippant, it is worth considering. However, many organizations do not have the freedom to change standards based on a technology choice. Fortunately, you have options; the first is to override the default CSS files using an ADF Faces feature called “skins”; the second is to make changes to the style properties for a specific component.

Working with Skins

An ADF skin offers a relatively easy way to make changes to the application. Instead of attaching CSS files to each page, ADF Faces defines a skin on an application level. An ADF Faces *skin* is a set of cascading style sheets containing selectors that define how each type of component will appear in an application. For example, a CSS selector is defined for the `af:inputText` component. Each time that component is rendered, it automatically uses that selector for its appearance. You do not need to apply the selector unless you want to override the default for a specific instance of a component.

Once a different skin is applied to the application, all components will then take on the characteristics defined in the new skin. Moreover, this change can be dynamic; that is, you can offer users the ability to change at runtime the skin that is being used to display the application. This ability has obvious benefits to users who are interested in specific color and font schemes for personal aesthetic reasons. It also has practical benefits for helping serve those who are visually disabled and whose interaction with the application could be easier with nonstandard fonts and colors.

You programmatically change from one skin to another by modifying a single application-level property. In the `ADFFacesTest` application you have used in this white paper, open `trinidad-config.xml` in the `WEB-INF` directory. You will see something like the following:

```
<?xml version="1.0" encoding="windows-1252"?>
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
  <skin-family>blafplus-rich</skin-family>
</trinidad-config>
```

The `skin-family` setting defines the skin. Change the “`blafplus-rich`” (or “`fusion`”) skin family value to “`simple`,” click Save All, and open a JSF page (for example, `dragDropDemo.jspx`) in the JDeveloper visual editor. (You may need to close the editor window and reopen the page for the changes to take effect.)

ADF Faces ships with three skins; the default skin “`blafplus-rich`” is modeled using the Oracle Browser Look and Feel (BLAF) standards; the effect of applying this skin and the other two skins—`blafplus-medium` and `simple`—to a simple JSF page with various components is shown in Figure 7. No changes were made to the page between screen shots; only the skin was changed. You will notice the minor difference between the `blafplus-rich` skin and the `blafplus-medium` skin—only the tabs look different in this set of components. However, the difference between the `simple` skin and the `blafplus-medium` skin is much more dramatic.

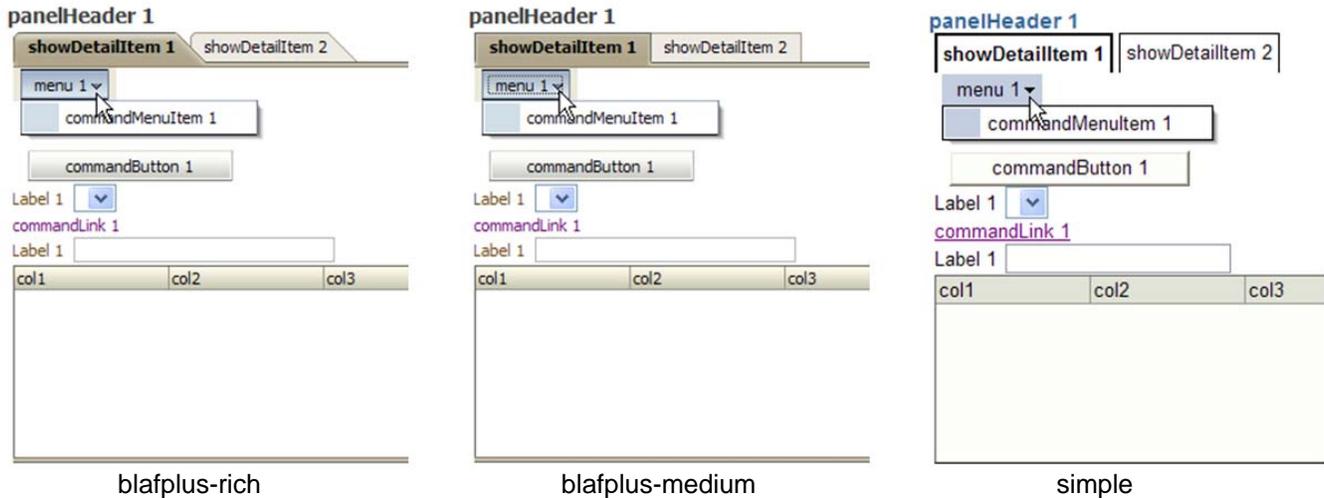


Figure 7. Different skins applied to the same page

If you want to try applying different skins and rerunning this same JSF page, drop components into a new JSF page called “skinsDemo.jspx” (no managed bean) so that they form the following arrangement in the Structure window. Nodes that are not expanded are empty. No properties were changed on any component after dropping them onto the page except the `af:spacer`, whose *Height* was set to “50.” Remember that `af:menuBar` appears as “Panel Menu Bar.”

Note

The screenshots in this white paper match the first full production version of JDeveloper 11g (build 5407) but in later releases of ADF Faces, the default look-and-feel (skin) may change slightly. This small difference is not important to your work while learning about ADF Faces.

Oracle Browser Look and Feel (BLAF)

If you need to create a skin, it is helpful to take inspiration from a fully developed look and feel. The Oracle Browser Look and Feel (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 Oracle EBS modules that present web browser interfaces. BLAF standards are exhaustively documented in hundreds of pages of public web files (www.oracle.com/technology/tech/blaf).

Tip

Even if you do not use BLAF standards, it is worthwhile using them for ideas about what type of look-and-feel standards to create and how the standards for different elements must complement each other.

Getting Started Defining Custom Skins

If you decide not to use the default ADF Faces skins, you can develop your own skin. The process is well documented in the JDeveloper help system. Enter “skins” in the help system search field and find the topic “Customizing the Appearance Using Styles and Skins.” This topic will guide you through the steps you need to perform to create a custom skin. 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 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 changes to the design while coding the skin. Other major steps follow:

1. Starting with the simple skin, 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.

Tip

You can extend an existing skin with your own skin, which means you can avoid writing an entire skin and, therefore, save time and effort.

2. Register the skin in a file called “trinidad-skins.xml” in the WEB-INF directory so that it is available to the *skin-family* property in the trinidad-config.xml file.
3. Activate the new skin by changing the skin-family property as described before.
4. Deploy the custom skin in a JAR file so that it can be more easily shared between applications.

Working with Component-Level Styles

A general development best practice is to take advantage of reuse and inheritance as much as possible so that maintenance and enhancements are easier; also, reuse theoretically means that you are taking advantage of pre-built and pre-tested code so fewer software defects should result. The ADF Faces use of skins falls into the category of reuse and inheritance because an application applies a consistent and tested appearance to each component.

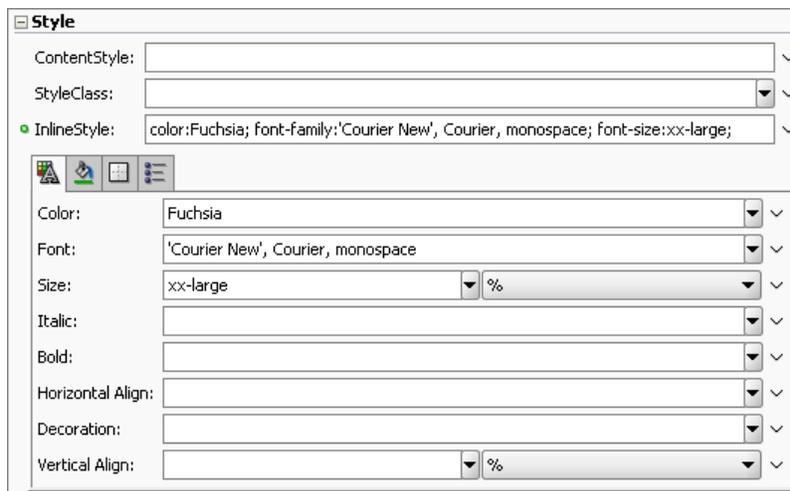
However, there are always exceptions to any rule, so it is useful to know how to override the styles set by the skin you are using. It turns out that each ADF Faces visual component offers a number of properties that you can use to alter the default appearance of a component without having to modify a style sheet. You have used some of these already, so the following is intended as a refresher.

Tip

If you find yourself overriding the same visual properties in the same way on most uses of the same component type, you need to consider making that change permanent in a skin.

Inline Styles

The *Inline Style* property appears in the Style section of the Property Inspector for most visible ADF Faces components (including layout containers). The property value can consist of a number of attributes and values delimited with semicolons (“;”). JDeveloper provides an editor under the *Inline Style* property that allows you to enter values for the *Inline Style* properties. The values and property names are dynamically loaded from the editor into the *Inline Style* property as shown here:



Modifying *Inline Style* properties uses the same techniques as other Property Inspector properties—a combination of typing values, selecting from a pulldown, interacting with a dedicated property editor (the Edit selection of the pulldown), or using the Expression Builder.

Caution

Although JDeveloper displays all *Inline Style* properties for all visual components, not all of these properties are appropriate for all component types. Rather than issuing an error message if an inappropriate property is set, ADF Faces just ignores the property. For example, a border setting is not applicable to an anchor (<a>) tag.

The properties in the Inline Style editor are divided into tabs for Text, Background, Box (for the area taken up by the component, most appropriate for layout containers), and Classification (for lists).

Tip

Be judicious when setting *Inline Style* properties, especially for nested layout containers, because a setting on a parent layout component can affect child components. If a component is not being arranged the way you expect, it could be due to an *Inline Style* property setting on a parent component.

Content Style

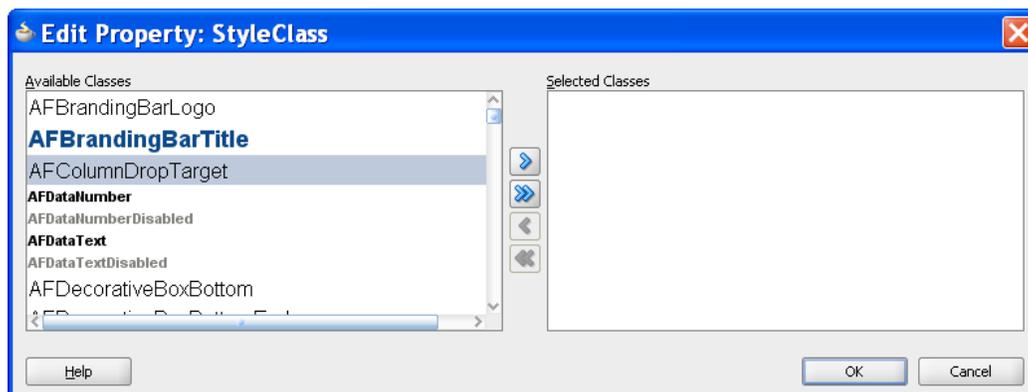
The *Content Style* property (shown in the preceding illustration) is available on components with data and other types of display such as `af:inputText`, `af:inputDate`, and `af:selectOneChoice`. For example, *Content Style* for the `af:inputText` component affects the way data in the field is displayed. If you set a *Content Style* width for an Input Text component, that width will override the *Columns* property of that field. The *Content Style* expression is the same format as the *Inline Style* expression.

Tip

You can use the Inline Style editor (tabbed property list in the Style section) to create an expression that you can cut and paste into the *Content Style* property. This editor makes it easy to create correctly formatted style expressions.

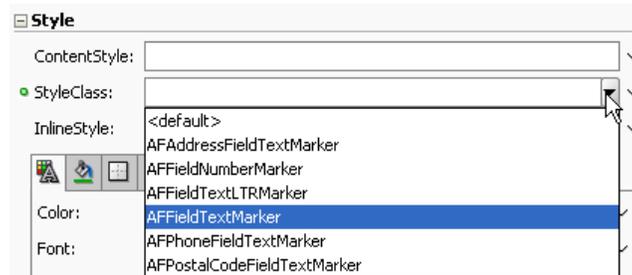
Style Class

The *Style Class* property (also shown in the preceding illustration) allows you to assign one or more preexisting style sheet selectors to the component. Selecting Edit in this property's pulldown displays the Edit Property: StyleClass dialog shown here:



When you move one or more selector classes to the *Selected Classes* area and click OK, the property value will be populated with the selected names. Selecting more than one class will apply attributes from all classes to the component.

You can enable ADF component styles to appear in a *Style Class* property pulldown by selecting the *ADF Faces Extension* checkbox in the Preferences dialog (**Tools | Preferences**) CSS Editor page. Then, when you select an ADF Faces component, the *Style Class* property will display a pulldown from which you can choose a selector specific to that component as shown here:



How Will I Ever Learn about All These Components?

Now that you've gotten a taste for some of the commonly used components and features, you may be feeling a bit overwhelmed because in addition to the ADF Faces techniques and components that this white paper introduces, you have gotten the sense that there are many more.

The key principle for further study is the same as always and is best expressed by Samuel Johnson (1820–1784) who said, "Knowledge is of two kinds: we know a subject ourselves, or we know where we can find information upon it." The following list several useful sources for finding the ADF Faces information you do not know yourself.

- **The JDeveloper Help Center** Mentioned before in this white paper is a single topic you can access by searching for "enhanced tag doc." This topic shows all visual components and provides links to detailed information about each component. Since this source is the closest to your work in JDeveloper, it might also be your first resort.
- **ADF Faces RC website** This Oracle Technology Network (OTN) page (www.oracle.com/technology/products/adf/adffaces) is the jumping off point for many resources related to ADF Faces. Many of the following Oracle Technology Network (OTN) pages are accessible from links on the first resource.
- **The Web UI Developer's Guide for Oracle ADF** This document (over 1200 PDF pages) is available in a downloadable PDF format from a link at the ADF Faces RC website just mentioned. This book is also part of the JDeveloper help system. Look for it under the Contents tree node Designing and Developing Applications/Developing Oracle ADF Applications. The website version may be more up-to-date than the version in the JDeveloper help system.
- **Tutorials, demos, and downloads** The ADF Faces RC website contains links to these information sources.
- **Cheat sheets** Look for the "ADF Faces Cheat Sheets" link on the ADF Faces RC website. The cheat sheets are intended to assist you selecting the correct ADF Faces component; they also provide details about some of the features for various components.
- **ADF Faces Rich Client Demonstration** This application is available from a link ("ADF Faces Rich Client Components Hosted Demo") at the ADF Faces RC website. You can run it on an Oracle-hosted server or download the application and run it locally. This demonstration contains a demonstration of all ADF Faces components; a separate demo application exists for DVT components. It also contains tips and advice on using ADF Faces components.
- **rea.oracle.com** The Rich Enterprise Applications (REA) website is available from the ADF Faces RC website. Look for the link called "Oracle REA." This website contains the hosted demo just described (click Explore Now, then expand Sandbox, and click Rich Client) as well as much more information about developing highly-interactive applications with products like Oracle Web Center.

- **AMIS ADF Blog** This consulting company, based in The Netherlands, is a frequent contributor to the ADF knowledgebase. Start searching for a topic at their ADF & JHeadstart home page (technology.amis.nl/blog/category/oracle/adf-jheadstart).
- **ADF EMG** The ADF Enterprise Methodology Group (groups.google.com/group/adf-methodology) discusses best practices for techniques and methodologies related to ADF. The prerequisite of experts contributing to this effort is that they know what “ADF” stands for.

Tip

After you drop a component onto a page, press F1 to display the tag help page for that component. This reference page is helpful for learning more about the component because it explains the purpose of the tag as well as its properties. For layout containers, it also lists the supported facets.

Summary

This white paper has introduced ADF Faces Rich Client and attempted to answer how rich (highly-interactive, flexible, and easy to use) it is by explaining the evolution of ADF Faces from UIX and early attempts to use AJAX-technologies to make the user interfaces more responsive and productive. The white paper also explained PPR—one of the main features that places ADF Faces RC on the top of the scale of interactive component sets. It also offered demonstrations in the form of hands-on practices for exploring features such as layout components, atomic components including the DVT, and skins. Lastly, it gave some direction for further study. These introductions and practices guides should help you be as productive as possible when developing your own richly interactive applications using ADF Faces RC.

*The rich are different from you and me.
They have more money.*

—F. Scott Fitzgerald (1896–1940), author,
and Mary McGuire Colum (1884–1957), critic

About the Author

Note: This white paper is based on material written by the author for the book *Oracle JDeveloper 11g Handbook*, Oracle Press, McGraw-Hill/Osborne (2009).

Peter Koletzke is technical director and principal instructor for the Enterprise e-Commerce Solutions practice at Quovera, in Mountain View, California, and has 26 years of industry experience. Peter has presented at various Oracle users group conferences more than 270 times and has won awards such as Pinnacle Publishing's Technical Achievement, Oracle Development Tools Users Group (ODTUG) Editor's Choice (twice), ODTUG Best Speaker, ECO/SEOUC Oracle Designer Award, ODTUG Volunteer of the Year, and NYOUG Editor's Choice (twice). He is an Oracle Certified Master, Oracle ACE Director, and coauthor of eight Oracle Press Books about Oracle development tools the latest of which are: *Oracle JDeveloper 11g Handbook* (with Duncan Mills and Avrom Roy-Faderman) and *Oracle JDeveloper 10g for Forms & PL/SQL Developers* (with Duncan).