GOING MOBILE: MOBILE APPLICATION DESIGN PRINCIPLES FOR ADF DEVELOPMENT

By Peter Koletzke, Quovera

I'm goin' home
And when I wanna go home
I'm goin' mobile
Well, I'm gonna find a home
And we'll see how it feels
Goin' mobile
Keep me movin'

—Pete Townshend (1945–),
Going Mobile (from the *Who's Next* album)

In the not too distant past, application developers and designers were often tasked with building applications that supported different resolutions in a desktop computing environment. Today they need to think about supporting mobile as well as desktop devices. User interface design for mobile devices requires more planning than user interface design for desktop computers because it needs to accommodate more than just different resolutions; specifically, it requires careful thought directed towards mobile device features such as touch interfaces, the ability to shift orientations, and embedded services.

Oracle has spent much time, money, and talent on the question of how to best design an application to run on mobile devices. They have worked much of this into the current version of Oracle Cloud Applications (f.k.a., Fusion Applications), and recently announced a new design system, Oracle Alta UI (Alta), aimed at mobile device support. Alta is guiding the design of Oracle cloud interfaces even outside the applications.

An Alta design includes the use of some relatively new user interface (UI) controls—more properly called "components"—now available to developers using Oracle Application Development Framework (ADF) in JDeveloper and Eclipse. (See the sidebar "A Word About ADF" for more about current use of the term "ADF.") These components fulfill current trends in mobile device interfaces.

This white paper presents a brief review of the evolution of user interface design for applications developed using Oracle tools. It then explores considerations for designing applications that are mobile aware and describes and shows an example of a common part of any application design—a navigation system or menu—and how its design will change for various user interface devices.

This white paper describes and shows examples of three ADF components that are relatively central to interfaces designed using Alta principles so you can understand why these components are not merely more tools in an already crowded toolbox—they are manifestations of the need to go mobile as well as the means to do so.

The descriptions and pictures in the first part of this white paper can go only so far. So that you can experience how these three components act and work, this white paper points to a sample test drive application that demonstrates these components. A note box at the end of this white paper describes how to download and run this application in JDeveloper 12*c*. In addition, a hands-on practice to develop this application is available in case you would also like to taste the development experience. Refer to the "Sample Application" note box in the conclusion of this white paper for details.

A Word About ADF

In this white paper as well as in current, Oracle-world parlance, the term "ADF" is used to refer to the technologies specific to Application Development Framework (for example, ADF Business Components, ADF Model, and in this situation, ADF Faces). Moreover, now that applications with most ADF technologies can be developed in either JDeveloper or Eclipse, when you work with ADF, you are considered an "ADF developer" rather than a "JDeveloper developer" or "Eclipse developer."

Oracle UIs Past to Present

For many years, Oracle Applications, Oracle's business applications suite, used Oracle Forms as its main user interface—first in terminal-server mode, then in client/server mode, finally in a web browser running a Java applet. After Oracle Applications was reworked into Oracle E-Business Suite (EBS), day-to-day data input screens built in Oracle Forms were supplemented with lightweight self-service, web browser, no-Java-applet-required screens using a proprietary Oracle technology: *UIX*—loosely expanded into "User Interface XML" (within which "XML" stands for "Extensible Markup Language").

In EBS, Oracle Forms still delivered on the requirements of fast data input and rich display needed for screens like sales order entry because web browser components at that time were not capable of that kind of productivity. Still, UIX supplied Forms-like functionality to the self-service screens through many innovative, highly-interactive components. An example of these components is a table control whose contents update without refreshing the entire page when the user scrolls up or down or sorts records by clicking the column headings. Although these components were on the industry's cutting edge, they only came close to but could still not match the interactivity users had become accustomed to in Oracle Forms applications.

When the JavaServer Faces (JSF) standard became mainstream, Oracle had already migrated its UIX components to be fully compliant with the JSF specifications. This component set, named "ADF Faces" and first available with JDeveloper/ADF 10*g*, increased the number of rich, interactive components available for web interface development. It also offered *declarative AJAX* (also called "partial page rendering"), which allows ADF developers to set up fast round-trip server events using only component property settings. (Refer to the next sidebar, "About AJAX" for an overview of AJAX.) With declarative AJAX, the developer is freed from the normal AJAX development burden of coding custom JavaScript functions and servlet code.

About AJAX

AJAX (or "Ajax") stands for "Asynchronous JavaScript and XML," and describes a combination of techniques that allow components on the web page to communicate with the server in the background. A summary of typical AJAX steps follows:

1. A JavaScript function creates an XMLHttpRequest object and uses it to send an XML message to the server; for example, after a user enters a postal code, the function would send an XML message containing the postal code to the server.
2. A custom servlet processes the request and sends a message back to the page; for example, the servlet would query a table to find the postal code and would return the associated city, state or province, and country.
3. Another JavaScript *callback* function accepts the values from the return message and places them in the proper locations on the page; for example, the function would fill in the city, state or province, and country fields.

The benefits of AJAX events are the small programming footprint, speed, and interactivity. That is, a small message is sent to the server; the server runs a very specific, and hopefully fast, program to form a return value; and the return value fills the values into one or more components on the page without redrawing the rest of the page.

ADF Faces in ADF (JDeveloper) 10*g* begat ADF Faces Rich Client (ADF Faces RC) in ADF 11*g*. ADF Faces RC (hereafter shortened to "ADF Faces") introduced even more components as well as extremely rich visual components like graphs, maps, and hierarchy viewers. Declarative AJAX moved forward even more with this version, and these components are now being used in Oracle's newest flagship application product, Oracle Fusion Applications ("Fusion Apps").

Although Oracle is using ADF 11*g* to develop Fusion Apps, it is also adding features to a new version: ADF 12*c* (released in July 2013). Some of these features, such as the components in this white paper's test drive application, are being ported back and forth between 11*g* and 12*g* versions so both versions can offer the new functionality.

Oracle's Mobile UI Tools

Oracle is investing heavily in user experience (UX) research and design primarily for Fusion Applications; lately, much of the UX effort has been on design for mobile devices. As a result, ADF has evolved again to solve the mobile requirement by offering the following alternative user interface technologies, each of which uses a separate set of components:

- **ADF Faces components**: used for desktop, tablet, and touch devices. It is very strong on network connectivity (to and from a remote database for example). Current version: 12*c* (12.1.3).
- **Mobile Application Framework (MAF) components**: used for Android and iOS mobile phones; this option also supports tablets and allows the application to access services on the device such as the camera, contacts, and GPS. It is not as strong as ADF Faces for network connectivity and may use a SQLite database local to the device that will be synchronized with a remote database. Current version: 11*g* (11.1.2.4).

Although this white paper concentrates on the first option, ADF Faces components, the available components, development techniques, and resulting user interface are similar for the second option.

A Change in Human Interaction Methods

Regardless of the ADF method, the application developer/designer needs to accommodate the shift in the way humans interact with the application. This has always been the case when interfaces changed.

For example, in the days of terminal-server, character mode applications, character and function key presses were the only way to input data and navigate an application. When graphical user interfaces (GUIs) appeared on the scene, application designers needed to consider the effects of mouse clicks and mouse movements in addition to keyboard key presses. Applications could now contain graphical elements like buttons that users could click to perform a certain function.

Mobile applications require another shift in design thinking because the human interface now favors hand gestures (finger swipes and taps) rather than mouse and keyboard actions. Most mobile devices also offer on-screen keyboard interfaces so mobile applications can use a combination of "traditional" typing and "modern" hand gestures.

In addition to the paradigm shift for human interfaces, the application designer can be faced with a requirement that the application support multiple device types: desktop computer and tablet, for example. However, the newer components, especially the newer components in ADF Faces, are built to support multiple device types: for example, a mouse click on an icon in a desktop environment is the same as a finger tap on the same icon in a tablet environment.

Design Shift Example

Think about how the design of an application menu has changed over the years based on the current human interface. In the days of character-mode applications a menu would often display as a list of functions with a number or letter next to each; a single field would allow input of the number or letter, and submitting that page would navigate to the function indicated in the field.

In GUI days, a menu could have been a menu bar with pulldown selections, or a navigation bar with multiple icons or buttons; selecting or clicking a menu selection, icon, or button would navigate the application to the selected function.

In mobile days, a menu can be rendered as a set of icons indicating available functions, which could look something like the screenshot on the right.

This view of options is particularly familiar to mobile "smart" phone users, who often see the available apps on their cellphone arranged in this way. In a desktop environment, the user would click one of the icons with a mouse to activate the selection; in a tablet environment, the user would finger tap an icon to activate the selection.

Once the selection is made (for example, Employees), the Employees page loads and the icons rearrange themselves at the top of the page as shown in Figure 1. The selected function is highlighted but the other functions are available for selection at any time, somewhat like a tab component.

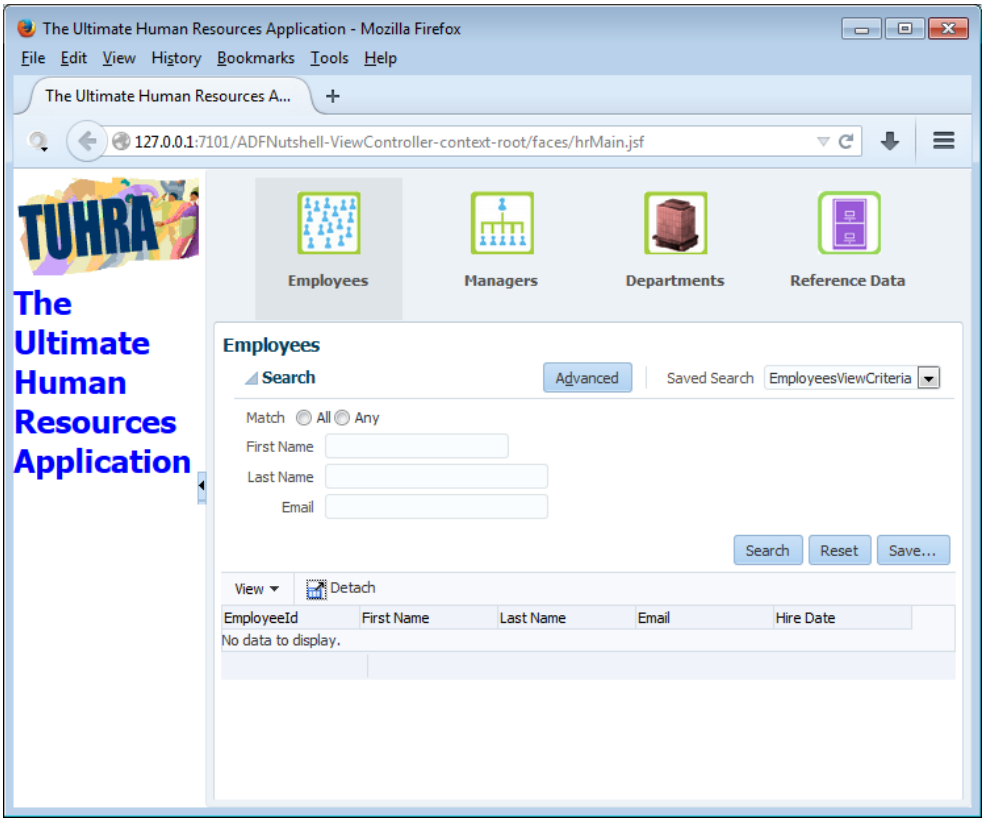
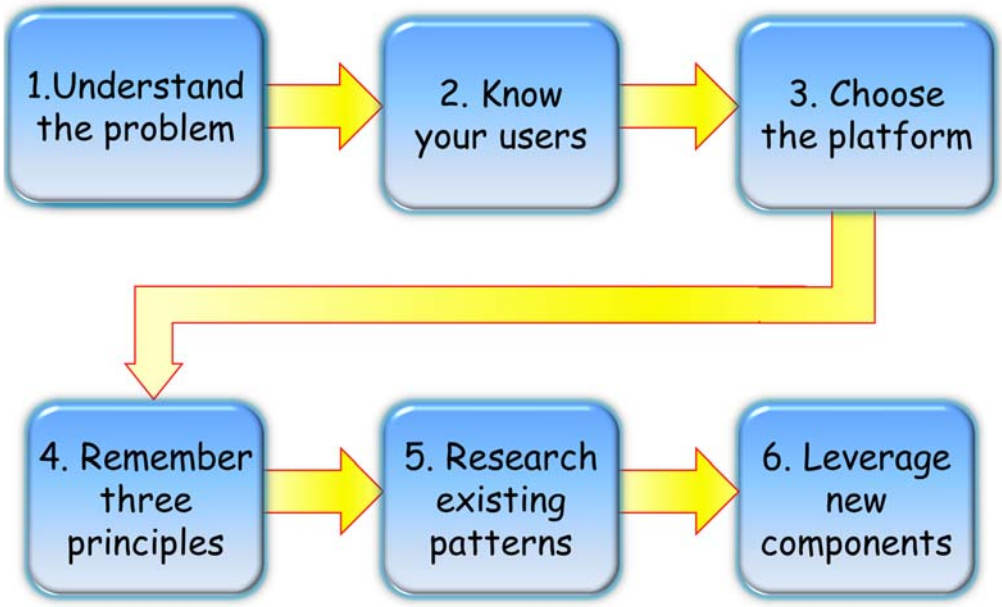


Figure 1. Employees page

Design Steps

While researching for this white paper, this author found that thoughts in the industry about designing applications for mobile devices fell into a number of categories that can be summarized in the following process for designing for mobile devices.



The discussion about mobile design principles can be organized around these steps.

1. Understand the Problem

This step entails recognizing two main factors: mobile is hot, and mobile is different.

Mobile is Hot

"Mobile" includes small portable devices of all types: smartphones, tables, and e-readers. Since there is a wide range of mobile products, you will not be surprised or maybe already know that the number of mobile devices in the world exceeded the world population in 2014 and that Apple Inc.'s 2014 Q4 revenue for the iPhone alone surpassed the combined revenue from all sales in Microsoft Corp. and Google. (http://www.bizjournals.com/prnewswire/press_releases/2014/10/06/NY30877)

This means it is probably inevitable that application developers will be faced with a requirement for supporting these devices.

Mobile is Different

The discussion before mentioned the difference in mobile device human interaction methods: tap and swipe. This means a mobile application can no longer rely heavily on a keyboard and mouse for input from the user. In addition, although desktop applications have been designed for accessibility in the past, and this included voice commands, with mobile voice is a more universal method for interacting with the device and needs to be supported in mobile applications.

Other than interaction methods, mobile display size is a big difference from legacy desktop applications. Therefore, mobile design needs to accommodate not only a smaller display device, but varying sizes of display devices. For example, smartphone displays may range from a couple of inches to six or more, while tablets and e-readers start around that figure and move up to 10 or 12 inches.

Another difference with desktop devices is that mobile devices are capable of shifting the display from portrait to landscape so the application must support that shift.

In addition, mobile devices come packed with features such as a camera, global positioning system (GPS), accelerometer, compass, and even a telephone. Mobile applications need to leverage these features.

2. Know Your Users

Understanding users has always been a design factor, unlike desktop applications that were run from…well, a desktop…with mobile, you need to know where the users will be while running the application. For example, will they be on the sidewalk or in a restaurant speaking with a customer, on a train, in a car, or somewhere else? Generally, you can count on the attention span for an application being much shorter than with a desktop application.

Alternatively, users may actually be working at a desktop, in which case they would be running the application using a desktop computer or tablet.

The reason this is important is that it allows you to predict the screen real estate available for the application as well as what your application can support. This leads into the next step.

3. Choose the Platform

Once you know where users will be when running the application, you can select the toolset you will use to build the application. As mentioned before Oracle supplies two main toolsets for mobile applications: ADF Faces for desktops and tablets, and MAF for smartphones and other small mobile devices.

4. Remember Three Main Principles

This step groups all principles into three main guidelines.

Design for Mobile First

Mobile applications can usually run without an issue on a desktop/tablet device. However, the traditionally-designed desktop application will display poorly at best on a mobile device. At worst, the application will not be useable at all.

Therefore this guideline is to design for the mobile device first, for example a tablet; then the application will work on the tablet and, since the desktop display is larger, it will also work on the desktop.

Designing for mobile also means using the new gesture- and orientation-aware components that accommodate both keyboard/mouse interactions as well as tap/swipe interactions. You will also want to think about taking advantage of the mobile services; for example, if the device you are supporting is a smartphone, you can use the GPS and camera.

Leveraging the devices connection to social collaboration features can also be a powerful tool in the application. The main guideline that relates to social collaboration is to remember that it exists and think about using it.

Visual, Not Textual

This guideline is a key success factor because the bar has been set by existing mobile applications to which your users might be accustomed. It is important to use simple design elements; for example, Alta offers an ADF skin that reduces the previously complex use of graphics to render multiple images for a single element like a tab folder. The Alta skin is built to be quick to render, flat in appearance (few gradients or shadows), and easy to read.

In addition, using large icons and color as emphasis can help transmit information that was previously textual. For example, a large circle can represent an amount of data that is more than a small circle; what is lost from the textual representation of those data items would be actual amounts, which could be transmitted as a mouse-over popup, or drilldown, or which may not be important.

Emphasizing the most important content is also an important design consideration. You can implement emphasis with special placement or by size or color.

Simple Navigation

Simple navigation is another key factor in design. It should be easy to return to the application's opening page and the content (data elements displayed on the screen), not the "chrome" (graphics like tabs or buttons) should be the navigation element. The reason is that less screen real estate taken by navigation elements means more screen can be used for information. When the screen is small, this is a useful cutback.

Summarizing essential data and providing drilldowns to details is another technique that can help make the best use of a small display. Designing the screen so it scrolls vertically only can help the user navigate as well.

Related to data display is the way data is queried. Since keyboard and mouse are downplayed or not available, you ned to substitute alternative means like single field searches with pulldowns for values and voice input.

5. Research Existing Patterns

A large amount of research and thought has already been put into the challenges of designing applications for mobile devices, so this step is learning about what is out there so you can take advantage of a pre-invented wheel.

One good place to start is Oracle Alta UI whose home page is here: bit.ly/oraclealta.

Alta guidelines are organized into the two main toolsets: Alta for desktops (some screens refer to "browser UI" or "web UI"); and Alta for mobile. Both toolsets contain a wealth of information and best practices like:

- Style guides: custom color palettes, icons, fonts, skins
- Component guides
- Patterns for page design
- Examples

In addition, Alta offers a sample application that showcases its guidelines: jdevadf.oracle.com/workbetter/faces/index.jsf.

6. Leverage New Components

Components in Oracle's toolsets support multiple interface types. Figure 1 is a screenshot from this white paper's test drive application. This application showcases these three new components that were designed to take Oracle's user interface experience into the mobile realm:

- Panel Springboard
- Panel Drawer
- List View

Panel Springboard

This component, which appears in the source code as `af:panelSpringboard`, is responsible for the icon arrangement and rearrangement just mentioned. It is also available in ADF Mobile as `adfmf:springboard`. It holds another component,

`af:showDetailItem`, for each icon displayed in the grid. For example, the code to render the four icon menu within the springboard component is the following:

```
<af:panelSpringboard id="ps2" displayMode="grid">
  <af:showDetailItem text="Employees" id="sdi1" icon="/images/peopleOff.png">
    <af:region value="#{bindings.empflow1.regionModel}" id="r1"/>
  </af:showDetailItem>
  <af:showDetailItem text="Managers" id="sdi2" icon="/images/managerOff.png">
    <af:region value="#{bindings.managerflow1.regionModel}" id="r2"/>
  </af:showDetailItem>
  <af:showDetailItem text="Departments" id="sdi3" icon="/images/buildingOff.png">
    <af:region value="#{bindings.deptflow1.regionModel}" id="r3"/>
  </af:showDetailItem>
  <af:showDetailItem text="Reference Data" id="sdi4" icon="/images/filecabOff.png">
    <af:region value="#{bindings.jobflow1.regionModel}" id="r4"/>
  </af:showDetailItem>
</af:panelSpringboard>
```

Notice that the displayMode property is set to "grid"; this defines the starting arrangement as a grouping of icons in the center of the display as shown in the first screenshot. You can alternatively set *displayMode* to "strip" to show the icons in a line as shown in Figure 1. Clicking an icon while the component is in grid mode will shift the display into strip mode automatically and will display the content in the `af:showDetailItem` as shown in Figure 1.

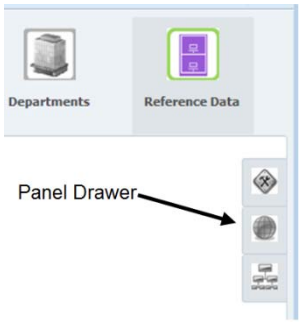
In the preceding code snippet, `af:region` defines a task flow that has been developed in another file. The task flows in this application are a single view (JSF page fragment) that displays data relevant to the selection, for example, Departments data for the deptflow1 flow.

To save space in this white paper, the Panel Springboard code has been abbreviated. Each `af:showItemDetail` can define separate icons for the "at rest" (initial) display, a mouse hover, disabled, and depressed (selected) as follows:

```
<af:showDetailItem text="Employees" id="sdi1" icon="/images/peopleOff.png"
                   depressedIcon="/images/peopleOn.png"
                   disabledIcon="/images/peopleOff.png"
                   hoverIcon="/images/peopleOn.png">
    <af:region value="#{bindings.empflow1.regionModel}" id="r1"/>
</af:showDetailItem>
```

Panel Drawer

The Panel Drawer, `af:panelDrawer`, component renders tabs attached to one side of its parent component. Clicking a tab opens the "drawer" and displays its contents. ADF Mobile has no exact equivalent but an ADF Mobile design could accomplish an equivalent effect with another embedded Panel Springboard.



The test drive application's Reference Data option displays a Panel Drawer as shown on the right. The tabs can appear on the left or right side of the parent container (or start or end side, if you are working with an application that needs to support multiple reading directions).

Clicking a tab in this component displays some content just as clicking an icon in the Panel Springboard displays content.

The code behind this component is very similar to the code behind the Panel Springboard:

```
<af:panelDrawer id="pd1" position="right">
  <af:showDetailItem text="Jobs" id="sdi5" icon="/images/toolsOff.png">
    <af:region value="#{bindings.jobflow1.regionModel}" id="r4"/>
  </af:showDetailItem>
  <af:showDetailItem text="Locations" id="sdi6" icon="/images/globeOff.png">
    <af:region value="#{bindings.countryLocflow1.regionModel}" id="r5"/>
  </af:showDetailItem>
```

```
    <af:showDetailItem text="Regions" id="sdi7" icon="/images/orgchartOff.png">
      <af:region value="#{bindings.regionflow1.regionModel}" id="r6"/>
    </af:showDetailItem>
  </af:panelDrawer>
```

Clicking a tab, like Jobs (the top icon), pulls out the drawer and displays the region task flow (JSF page fragment) inside the `af:showDetailItem` component as shown in Figure 2.

Subsequent tab selections retract the visible drawer and pull out the selected drawer. The `af:showItemDetail` inside this component offers the distinct icon properties mentioned before (at rest, mouse hover, disabled, and depressed).

With tabs on the right (or left) a tablet user can select an icon with the thumb on the hand holding the tablet. The size of the extended drawer will be the size of the initial content. If the content changes, the drawer will not resize but you can alternatively set width and height attributes to declare the size of the drawer.

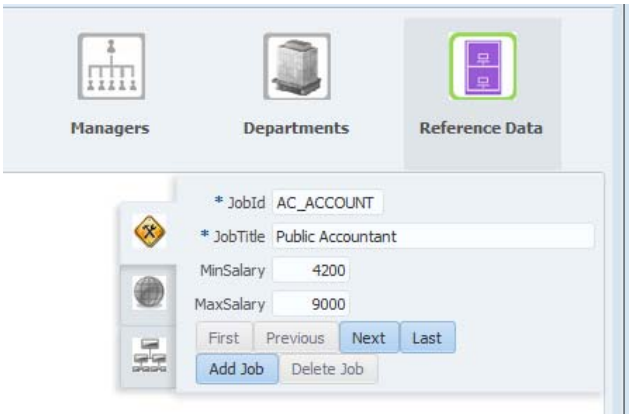
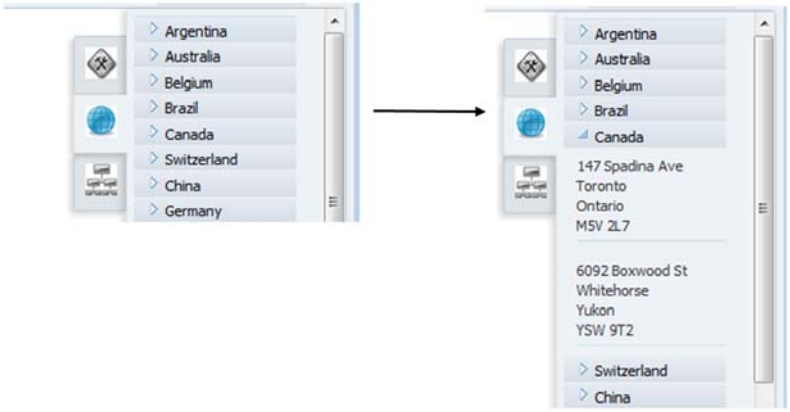


Figure 2. Extended Panel Drawer

List View

The List View component, `af:listView`, is responsible for displaying a collection of data (set of rows). It also supplies functionality familiar to smartphone users. That is clicking (or tapping) a selection in the list can reveal a panel with details about the item. It is available in ADF Mobile 11*g* as `adfmf:navigationBar`.

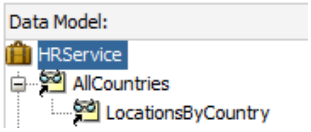
As a static demonstration, consider the following flow from the Locations page (second drawer of Reference Data) in the test drive application:



Clicking the show details “>” icon next to Canada, as in the snippet on the left, expands a panel under it showing the offices in Canada. The code to display this component looks like this:

```
  <af:listView value="#{bindings.AllCountries.treeModel}" var="item"
               emptyText="#{bindings.AllCountries.viewable ? 'No data to display.' :
```

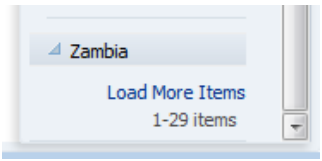
```
                    'Access Denied.'}"
            fetchSize="#{bindings.AllCountries.rangeSize}" id="lv1">
    <af:listItem id="li1">
      <af:panelGroupLayout id="pgl1" layout="vertical">
        <af:outputFormatted value="#{item.bindings.StreetAddress.inputValue}"
                            id="of1"/>
        <af:outputFormatted value="#{item.bindings.City.inputValue}"
                            id="of2"/>
        <af:outputFormatted value="#{item.bindings.StateProvince.inputValue}"
                            id="of3"/>
        <af:outputFormatted value="#{item.bindings.PostalCode.inputValue}"
                            id="of4"/>
        <af:separator id="s1"/>
      </af:panelGroupLayout>
    </af:listItem>
    <f:facet name="groupHeaderStamp">
      <af:listItem id="li2">
          <af:panelGroupLayout id="pgl2">
            <af:outputFormatted value="#{item.bindings.CountryName.inputValue}"
                                id="of5"/>
          </af:panelGroupLayout>
      </af:listItem>
    </f:facet>
  </af:listView>
```



Naturally, this display requires queries to the database, and view objects in the ADF Business Components (ADF BC) layer of the application define these queries. The data model of the ADF BC application module looks like the screenshot on the right for the queries used by the `af:listView`.

AllCountries represents a view object that queries the COUNTRIES table; LocationsByCountry defines a view object query on the LOCATIONS table and is joined to AllCountries with a view link (foreign key relationship) defined on the COUNTRY_ID column. Notice that the treeModel that supplies the value (data) to the `af:listView` is based on AllCountries. The page definition binding for AllCountries contains an accessor based on the view link, which allows the master, AllCountries, to access the detail, LocationsByCountry, attributes.

Since List View is a multi-row (collection) object, it uses a tree binding with nodes for the country name and locations attributes. The component's `groupHeaderStamp` facet contains an `af:listItem` that displays the country name. The component's direct child component, `af:listItem`, displays the location information. The example in this application uses read-only, `af:outputFormatted`, components, but it could just as well use `af:inputText` components if data entry was required.



Notice from the flow screenshot that the component automatically renders a scrollbar to navigate through the list returned by the AllCountries query. It also automatically displays a "Load More Items" selection such as that on the right if the scrollbar cannot cover the entire list returned from the query.

This component can be used to find a value from a list and display details as in the preceding example. It can also be used as a navigation list: so when a value from the list is selected, appropriate data is selected in adjacent regions. The test drive application contains an example of this use in its Managers page as shown in Figure 3.

Selecting a manager's name on the left queries the manager's record on the right and under that record is a read-only list of employees reporting to that manager. The List View component in this example is simpler because it only needs to display the manager's name, without any "show detail" expanding panel functionality. The List View code looks like this:

```
  <af:listView value="#{bindings.AllManagers.collectionModel}" var="item"
            emptyText="#{bindings.AllManagers.viewable ? 'No data to display.' :
                      'Access Denied.'}"
```

```
            fetchSize="#{bindings.AllManagers.rangeSize}" id="lv1"
                      selection="single"
            selectionListener="#{bindings.AllManagers.treeModel.makeCurrent}">
    <af:listItem id="li1">
      <af:panelGroupLayout id="pgl1">
        <af:outputFormatted value="#{item.bindings.FirstName.inputValue}" id="of1"/>
        <af:spacer width="7" height="10" id="s1"/>
        <af:outputFormatted value="#{item.bindings.LastName.inputValue}" id="of2"/>
      </af:panelGroupLayout>
    </af:listItem>
  </af:listView>
```

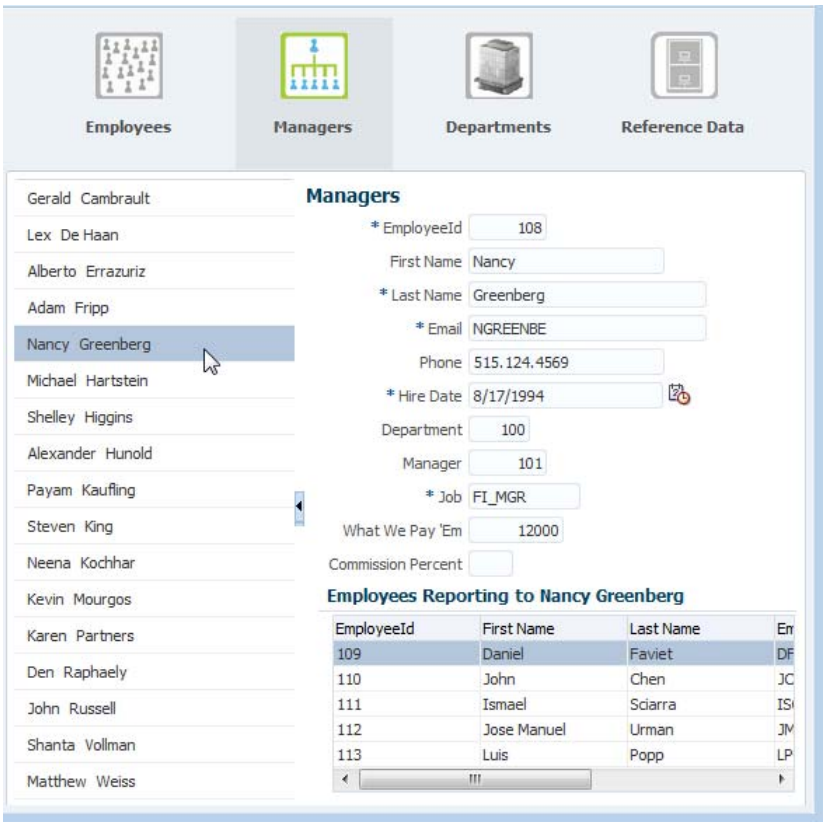


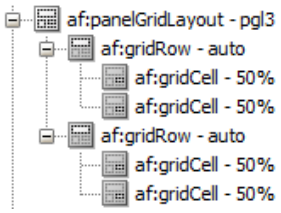
Figure 3. Managers page

The data model for the List View component does not require a nested collection like the preceding country-location example. It is a simple list based on a single view object. However, the page includes a nested collection for the employee detail table. In addition, the page includes declarative AJAX (partial page rendering) settings for updating the form and table values on the right when a manager is selected on the left. Refer to the hands-on practice for implementation details.

Bonus Component: Panel Grid Layout

Although out of scope for this white paper's sample application, another important, new component was added with ADF 12*c* that can assist in solving design challenges in both desktop and mobile applications.

Panel Grid Layout is another choice in the group of *layout components*: a component that holds other components. Panel Grid Layout, `af:panelGridLayout`, allows you to specify a layout of a certain number of rows and columns, with the ability to use HTML-like properties such as column spans, row spans, and margins. It emulates the Java Swing GridBagLayout or GridLayout layout managers by using embedded grid rows and grid cells. For example, a simple two-row, two-column layout would look display as follows in the Structure Window:



Before this component was available, the grid functionality usually required embedding multiple layers of layout components, and this often resulted in inconsistent and sometimes unpredictable behavior. Panel Grid Layout greatly reduces the need for complex, multi-level layout component designs, and is now a highly-recommended layout component for most multi-cell needs.

Conclusion

This white paper has explained how Oracle user interface technologies have evolved and the benefits derived from each step in that evolution. It also explained how the relatively recent change in human interfaces has required a paradigm shift in designing applications. The white paper gave an example of this paradigm shift for a menu component. It then described the general steps you will follow when approaching design for an application that needs to support mobile devices.

The white paper provided some technical detail about the coding for three relatively new ADF user interface components: Panel Springboard, Panel Drawer, and List View. It also briefly mentioned the place of honor immediately taken by another new component: Panel Grid Layout.

The Next Step – a Test Drive

This stuff is kind of exciting! Even if you disagree with that bold statement, you may still want to try out the application mentioned in this white paper. The sidebar “Sample Application” contains details of where you can find it.

In addition, you may also want to try out some development work with these components, and the sample application file download includes a hands-on practice you can use to build the test drive application.

Running this test drive application and working on the hands-on practice will give you a good feel for how ADF handles the needs of a mobile application. With the overview in this white paper, the test drive, and hands-on practice, you will be equipped with some of the knowledge you will need should you find yourself goin’ mobile.

Sample Application

Conference attendees can find the test drive application and hands-on practice mentioned in this white paper on the conference website where the presentation slides and this white paper are also available. Alternatively, you can find these resources at www.quovera.com: click White Papers and look for the hands-on practice under “Oracle ADF On-Ramp: Developing Applications with the Core Oracle ADF Stack.”

About the Author

Peter Koletzke is a technical director and principal instructor for Quovera, in Palo Alto, California, and has over thirty years of industry experience. Peter has presented at various Oracle users group conferences more than 330 times and has won awards such as Pinnacle Publishing's Technical Achievement, Oracle Development Tools Users Group (ODTUG) Editor's Choice (three times including one for this white paper), ODTUG Best Speaker, ODTUG Volunteer of the Year, NYOUG Editor’s Choice (four times), and ECO/SEOUC Oracle Designer Award. He is an Oracle Certified Master, Oracle ACE Director, and coauthor (variously with Dr. Paul Dorsey, Avrom Roy-Faderman, and Duncan Mills) of eight Oracle Press development tools books including *Oracle JDeveloper 11g Handbook*.