JDEVELOPER 10*G* WITH THE FUSION STACK AND JHEADSTART– IS IT ORACLE FORMS YET?

Peter Koletzke, Quovera

> ***Laws of Serendipity***
>
> 1. *In order to discover anything, you must be looking for something.*
> 2. *If you wish to make an improved product, you must already be engaged in making an inferior one.*

Oracle has long offered tools that assist with developing applications; its current focus is with the Java IDE, JDeveloper. JDeveloper 10*g* (release 10.1.3) offers the Oracle Application Development Framework (ADF), a set of features, libraries, and methods, which greatly simplifies Java development tasks and takes advantage of the popular Java Platform, Enterprise Edition (Java EE, formerly known as "J2EE") technologies. Oracle is currently using ADF with a specific set of these technologies to create its next version of E-Business Suite, called "Fusion Applications." This technology set, or "Fusion Stack" is centered around Java compliant technologies such as JavaServer Faces (JSF), ADF Faces (Oracle's JSF component set), and ADF Business Components (ADF BC, a database communication framework).

Working within JDeveloper can be a primarily declarative and visual experience and in this way, developers can create Java EE web applications in an environment that is closer to Oracle Forms than any tool previously released.

This white paper describes JDeveloper, Application Development Framework (ADF), the Fusion Stack, and how they can assist with development of a Java-based application. It also discusses the development process used to create applications using these technologies. In addition, this paper provides an overview of the JHeadstart add-on and shows how its declarative environment can assist Forms developers even more. Finally, the paper draws some comparisons between Forms and JDeveloper, and offers insights about how a traditional Oracle development shop can benefit the most from the Fusion Stack and JHeadstart. The white paper closes by attempting to answer the question many are asking: is it really possible to achieve the productivity of an Oracle Forms environment by using the Fusion Stack (with or without JHeadstart)?

JDEVELOPER

JDeveloper is Oracle's integrated development environment (IDE) for creating code based around standards in the Java EE standards. JDeveloper has evolved from a its early releases based on Borland's JBuilder to its current 10.1.3.3 release, which supports development of all types of J2EE 1.4 code as well as some code using the newer Java EE (version 5.0) platform standards. JDeveloper is highly-acclaimed by Java industry journals, although still sorely underused outside of Oracle development communities. Its chief competitor is the open source IDE, Eclipse. JDeveloper is not an open source product, although it is offered for no charge to Java developers who do not use ADF for runtime. (ADF runtime is included as part of the Oracle Application Server license.)

> **Note**
>
> As of this writing, Oracle has not stated any date or range of dates for the release of Fusion Applications as well as for Oracle Application Server 11*g* and JDeveloper 11*g* used for Fusion Applications. However, a technical preview version of Oracle JDeveloper 11*g* is available from otn.oracle.com now. The JDeveloper 11*g* release is being used to create the Oracle Fusion Applications. The core Fusion Stack this white paper refers to is available now within JDeveloper 10*g* as well, although JDeveloper 11*g* adds support for additional technologies used in Fusion Applications.

THE FUSION VISION

Oracle Fusion is a strategic reorganization of Oracle products. Oracle embarked on the Fusion road after acquiring various companies who had their own application products. Oracle's objective with Fusion is to merge the best of all those products

into a single (Fusion) applications suite. This effort will take many years, but Oracle has started the work and has stated that their work will use the Fusion Middleware products including JDeveloper and Oracle Application Server to create the Fusion Applications. Therefore, Oracle is very focused on enabling JDeveloper to support all requirements of the new Fusion Applications.

A large percentage of Oracle's business is in their packaged applications. Therefore, it is in Oracle's best interest to make the tool they are using to create the new Fusion Applications—JDeveloper and the technologies they use within it for their Fusion applications—as fully featured, easy-to-use, and robust as possible. Oracle's use of their own tools to create their highly-popular applications is nothing new. Oracle developed its pre-Fusion applications (E-Business Suite) using Oracle Forms and Reports. That Oracle applications has relied on Oracle development tools in the past has been one of the driving factors in the success that Oracle customers have had in their use of the same tools for custom development.

Note

In addition to the Fusion technologies this white paper discusses, Fusion Applications will make heavy use of Service-Oriented Architecture (SOA), Enterprise Service Bus (ESB), and Business Process Execution Language (BPEL). All of these are part of the Fusion Stack for the purposes of Fusion Applications and will be well supported in JDeveloper 11*g*, but work outside of Fusion Applications can still use any or all of these technologies now in JDeveloper 10*g* and in the future with JDeveloper 11*g*.

APPLICATION DEVELOPMENT FRAMEWORK

The first version of Oracle JDeveloper 10*g* (release 9.0.5) introduced Oracle Application Development Framework and each subsequent JDeveloper release has further refined ADF. The splash screen of JDeveloper release 10*g* declares the motto of "Productivity with Choice." The choice comes from the many technologies, deployment platforms, and development styles that JDeveloper supports. The productivity results from the ADF development method and architecture.

In addition to feedback received from expert user testing and months of feedback from a public preview release available on Oracle Technology Network (OTN), ADF had been road tested over the course of four years by more than two thousand of Oracle's E-Business Suite application developers. From that standpoint, ADF is a mature product even though it was first released to the public with JDeveloper 10*g*.

ADF provides a common method of development for many types of code and integrates a number of popular Java EE frameworks. Since understanding the word "framework" is important to the understanding of ADF, it is good to start with an explanation of that term.

WHAT IS A FRAMEWORK?

The term *framework* is used in the Java world to refer to current application development technologies. A framework has some features in common with an Application Programming Interface (API) or a code library: all offer generically built code that you can use in your application. The code that implements the framework supplies an entire service that you can access using a certain development method. Although APIs and code libraries may or may not have these characteristics, frameworks are built around the idea of a service. For example, instead of building from scratch some key service such as a connection layer to the database, you use an existing framework to supply that service.

One reason to use a framework is that you are able to tap into a standard way of adding the functionality to your application. You do not need to reinvent the particular wheel that you need for a piece of your application. Another related reason is that you do not need to redevelop code that many applications have in common. You leverage solid and (hopefully) well-debugged code in all your applications. In addition, the most popular frameworks offer solid support at least from the user community, if not from a vendor.

The foundation of a framework is one or more code libraries that supply the required service. For example, ADF Business Components is an Oracle framework that offers object-relational (OR) mapping and high-level access to Java Database Connectivity (JDBC) functions; the base classes in the ADF BC library are written generically so you can tap into their functionality for your application.

Another framework example is Apache Struts, an open source project. Struts supplies the Controller layer of a web application, which manages page flow and the processing of user interface events such as button clicks. As with ADF BC, Struts base classes supply this service, and you define how it works for your application using XML files and extending the Java base class files, if necessary.

Framework Use of XML Files

The generic library code in a Java framework is usually customized for an application need by the use of code in an XML file. For example, when you develop ADF BC code, you develop small XML files that define how the ADF BC library code will run for your application. The ADF BC based classes offer generic and developer-friendly ways to access JDBC functions and to construct SQL for those functions. Using XML files, you indicate to the framework classes how your database tables and views are structured (column names, datatypes, constraints, and so on). Then, ADF BC constructs the correct SQL for your tables and handles return values and messages from the database.

For ADF BC development in JDeveloper, you create and edit XML using wizards and property editors. You interact with these editors in a declarative way and JDeveloper creates XML code from the property declarations. This style of development and the use of XML files are common to many Java frameworks. A guiding principle of frameworks is that most of the application-specific tweaking of the framework can and should be accomplished by using declarations in XML files, not by writing procedural code in a Java class file. This is good news for developers who are accustomed to the declarative style of programming such as that within Oracle Forms.

Framework Use of Java Files

Your application may (and most likely will) require features or behavior that the base framework classes with your specific XML declarations do not offer. Therefore, another guiding principle of Java frameworks is that they can be extended. That is, you can supplement or replace part of the service by writing extensions (subclasses) to the base framework classes. Methods that you write in these extensions are used automatically at runtime in addition to or instead of the methods in the base classes. Since your extensions subclass the base classes, the rich base class functionality will still be used for other work.

When working with frameworks, a best practice is to use this feature sparingly. The more code you write to extend the framework, the less transparent upgrades or patches to the base classes may be. In addition, you use frameworks to avoid having to write a lot of repetitive, complex code. Therefore, if you find yourself writing a lot of code to extend the framework, perhaps the framework is not a correct match for your needs.

ADF as a Meta-Framework

ADF provides a common development method and common set of tools that allow you to work with many different frameworks such as ADF BC and Struts. In this sense, we can think about ADF as a *meta-framework* —a framework that integrates other frameworks. Before looking at the tools that ADF offers to support integrated work with various frameworks, we need to obtain a taste for the various frameworks that ADF supports. The best way to understand these frameworks is by examining and briefly discussing the ADF architecture model.

ADF Architecture Model

ADF is divided into four layers that roughly follow the Java EE Model-View-Controller (MVC) design pattern. Figure 1 shows a representation of these layers and the ADF-supported technologies that fit into them. JDeveloper's design time tools can create code from the technologies shown in all ADF layers. You will notice separate technology tracks and communication paths defined in this diagram for two styles of code deployments: application client and web client.

The Java EE specifications define *application client* as code that runs inside a *Java Virtual Machine* (JVM or Java runtime) on the client machine. With application client, the developer writes the application mostly in Java. On the other side, *web client* is code that runs in a JVM on an application server. Many users share the same runtime and the user interface is most commonly rendered in a web browser on the client's desktop machine. The web client concept also supports uses for other shared server runtimes such as wireless (cellphone or PDA) and telnet (when Java code is used to generate user interface items in character mode). For a browser web client application, Developers use HTML and Java markup tag languages primarily, although Java library code is always present in an underlying layer (usually in pre-existing libraries).

About MVC

The MVC design pattern upon which ADF is built defines three main layers of application code:

- **Model** This layer represents the data and values portion of the application.
- **View** This layer represents the screen and user interface components.
- **Controller** This layer handles the user interface events that occur as the user interacts with the interface (view), controls page flow, and communicates with the Model layer.

In concept, these layers are independent so that you can switch code in one layer to another technology and use the other two layers without modification. For example, if you build your View layer based on JavaServer Pages (JSP) technology, you could switch that View layer for one built with JavaServer Faces (JSF) and retain the same Model and Controller layers. In practice, completely separating layers is difficult, but MVC is still useful as a guide and a goal.

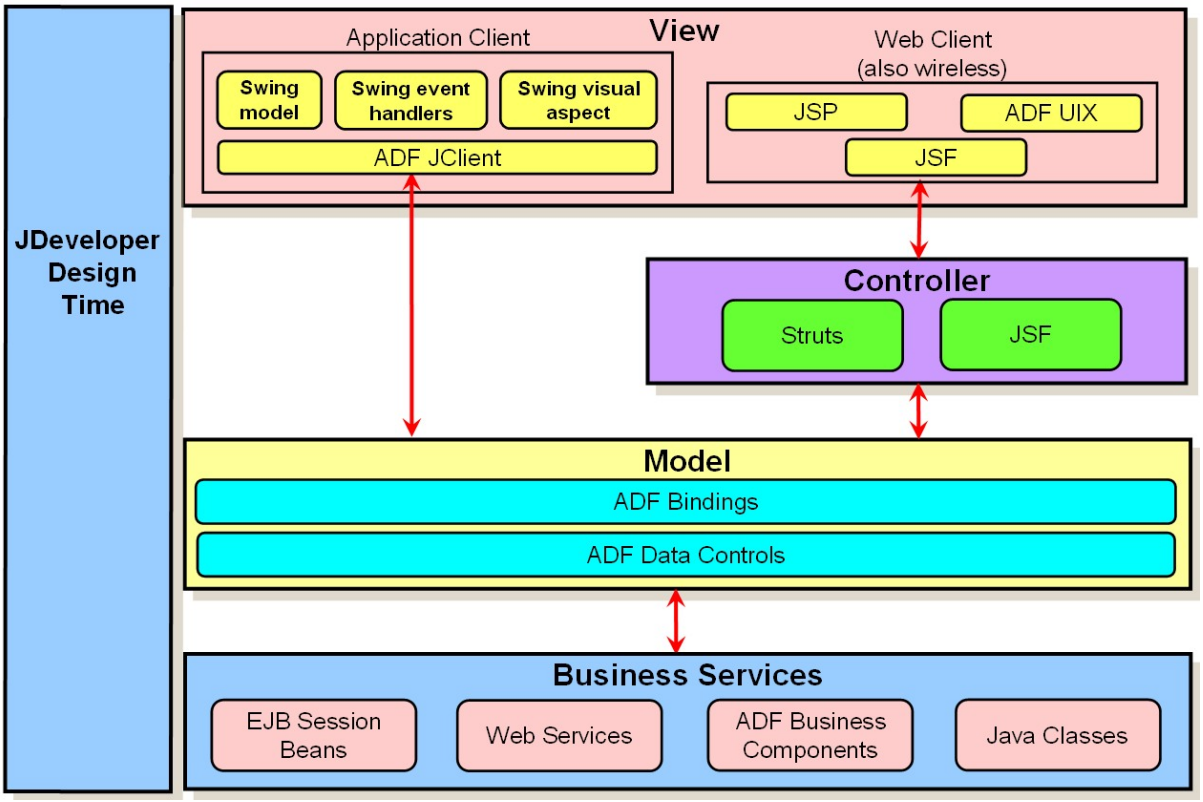


Figure 1. ADF architecture model

Business Services Layer

ADF adds Business Services layer (shown in Figure 1) to the layers of the MVC design pattern. This layer represents data sources and is actually a spin off of some functionality in the MVC Model layer. ADF excels and is unique in its handling of business services (data sources). Much thought and effort have gone into providing a seamless method for connecting the various layers of an application to its data. ADF handles the complexity of binding the user interface components to various sources of data. The binding is handled in the Model layer and the data sources are handled in the Business Services layer.

The Business Services layer of ADF provides code for accessing data sources such as a database. Business services are responsible for *persistence*—the physical storage of data for future retrieval—and *object-relational (OR) mapping*—translating physical storage units such as rows and columns in relational database tables to object-oriented structures such as arrays of objects with property values.

ADF was designed around the idea of flexibility. For example, if you are accustomed to working with business services delivered in Enterprise JavaBeans (EJBs), you can implement the ADF Business Services layer using EJBs. ADF will then seamlessly integrate this data access source into the other ADF layers.

ADF supports the following business services technologies:

- **EJB**—A standard Java EE structure for managing data from within a runtime container, consisting of entity beans, session beans, and message-driven beans.
- **Web services**—Utility functions and other resources written by a provider that are available through an Internet address and that you can incorporate into your application.
- **ADF Business Components**—Mentioned before, ADF BC is an evolution of Business Components for Java (BC4J), which provides components that allow developers to design and code business objects and business logic. It offers easy interaction with business data though SQL statements.
- **Java classes**—You can code Java class files, also called *plain old Java objects* (POJOs) or *JavaBeans*, that supply data from any location (for example, files, Java objects, or a database). JDeveloper's TopLink utilities provide flexible mapping and persistence services to these Java classes.

MODEL LAYER

The Model layer in ADF architecture (as shown in Figure 1) supplies the connection mechanism from the View layer to the data access components in the Business Services layer. It receives instructions from the Controller layer as requests for data retrieval and updating. The Model layer supplies data from the Business Services layer and sends a request to the View layer to update the display. For example, when the user submits a page in a web client application, the Controller layer requests an update of the data through the Model layer. The Model layer communicates the data change to the View layer so the visual display can be updated when the page refreshes (in the case of a web client).

The Model layer in ADF is composed of two aspects—ADF Data Controls and ADF Bindings. Before discussing these aspects, a brief explanation of data models is needed because data controls are defined for a data model supplied by a business service.

DATA MODEL AND VIEW OBJECT INSTANCES

A *data model* is a representation of the business service objects available to a project. For example, if you create a business services layer using ADF Business Components, you can define a data model in the application module definition using the Application Module Editor as shown in Figure 2.

The left area of this editor shows the available data objects and the right side shows the data model—objects selected for this application module. You can also use the business components diagram to modify or display the application module data model.

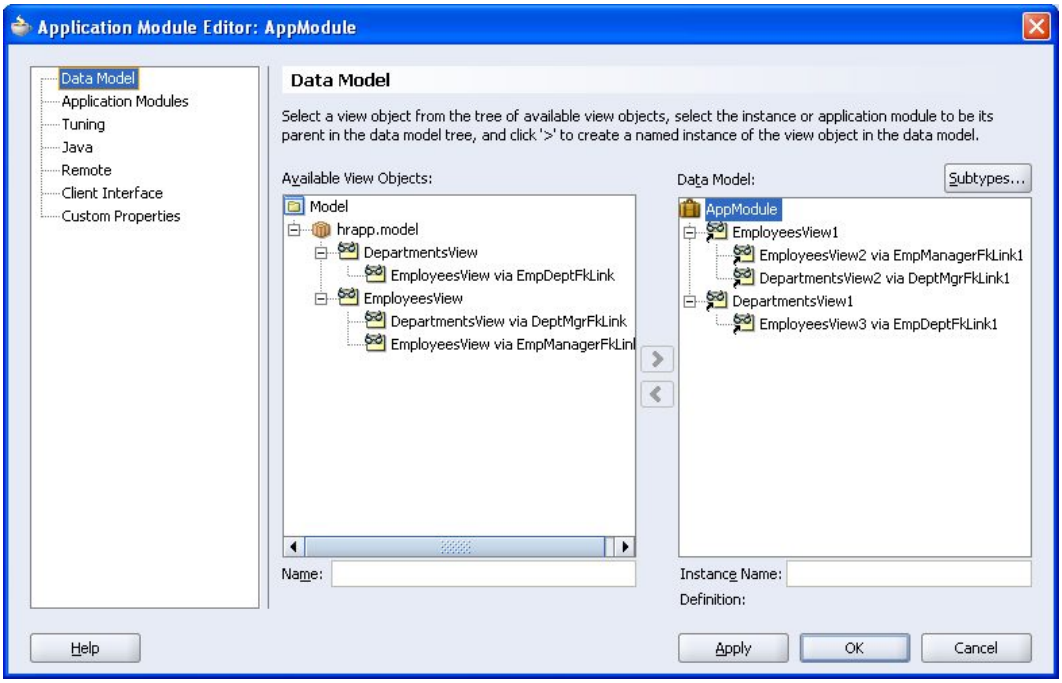


Figure 2. JDeveloper's Application Module Editor

This data model contains two master view object instances that represent database tables—EmployeesView1 and DepartmentsView1. It also contains detail view object instances such as EmployeesView3, which links to the master view object instances. The view object definitions and their attributes are available to the applications that reference the data model. The master-detail links represent the same type of master-detail link as a foreign key constraint in the database. For Forms developers, these links are like *relations* between Forms blocks—they synchronize the viewable detail records (for example, Employees) in the context of one row in the master (for example, Departments).

ADF DATA CONTROLS AND THE DATA CONTROL PALETTE

Data controls are built from the data model and are used to abstract one or more business services into a common layer. For example, JDeveloper projects that use different business services such as ADF BC, EJBs, and web services can access all of them using a single tool— the Data Control Palette (shown in Figure 3). Data controls appear as a list of data model components and you can drag them onto a user interface page to create UI components.

The Data Control Palette offers user interface controls that are appropriate to the kind of page or panel you are working on (for a web client or Java client, respectively). The interface controls presented also depend upon whether the data source is

a collection of data (such as an ADF view object instance with multiple rows), a single value (such as an attribute in a view object instance), or a single structure (such as a row). Different binding objects are supported depending upon the type of data and the controls available in the Data Control Palette will change based on these considerations.

Some of the user interface components available for view object-level components such as EmployeesView1 and DepartmentsView2 are Read-Only Table, Navigation Buttons, Input Form, Read-Only Form, Select Row Link.

Some data model components that are available for attributes include Value, Label, Text Field, Password Field, and List of Values.

Data controls are available from the Data Control Palette, which appears automatically when you display a visual editor (or when you select **View | Data Control Palette**). All data model objects defined for business services in the workspace will be available. When you drag and drop a data model object from this palette to a visual editor, a menu will appear that displays the list of available interface controls. After you select a component from this menu it will appear in the visual editor. Figure 4 shows this action for a Read-Only Table built from the EmployeesView3 detail view object definition.

In addition to data value components, the Data Control Palette also offers "operations" such as Commit and Rollback to send the current data in the Model to the Business Services layer (and, in the case of ADF BC, to the Oracle database). Other navigational operations such as Create, Find, First, Next, Last, and Delete appear on the data set (ADF BC view object definition) level.

ADF BINDINGS

Bindings are code or definitions that declare which data from a business service will be connected to a user interface control or structure. One of the challenges in the classic definition of MVC has been where to place the data-binding functionality. ADF Bindings act as the connection layer from the View components to Business Services components. Bindings are defined using a combination of property values containing expression language (EL) values that reference definitions in a PageDef XML file. These bindings are created and maintained automatically by actions in the JDeveloper IDE but are available for editing, if needed. The binding attaches the component to data from the business service at run time.

CONTROLLER

The Controller layer in ADF is used only for web client code as shown in Figure 1. The Controller layer defines *page flow*—which page is presented when an action occurs on another page—as well as the processing actions that occur between pages (such as a database query). Since the Controller handles the order in which pages appear, one page need not have a hard-coded link to the next page. This makes the page flow design more flexible because the Controller can apply conditional logic to determine the next page to be displayed. The Controller layer is also responsible for sending data entered in the View layer to the Model layer where it can be processed.

Although you can use any controller mechanism (even one you write yourself) for web client code you create in JDeveloper, the tools in the current release of JDeveloper 10*g* support the Struts framework mentioned earlier. It also supports the JSF controller; this controller is bundled with user interface (View layer) components inside the JSF framework (explained later in this paper).

VIEW

The View layer in ADF (shown in Figure 1) includes application client and web client technologies that are used to render user interfaces.

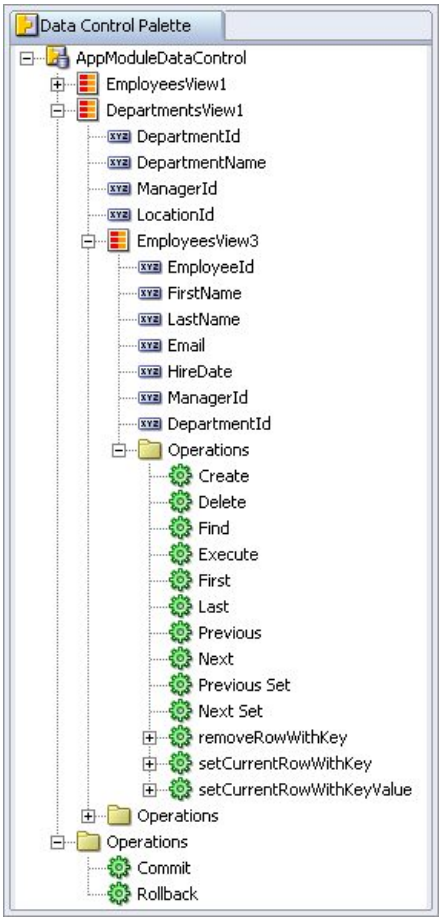


Figure 3. Data Control Palette

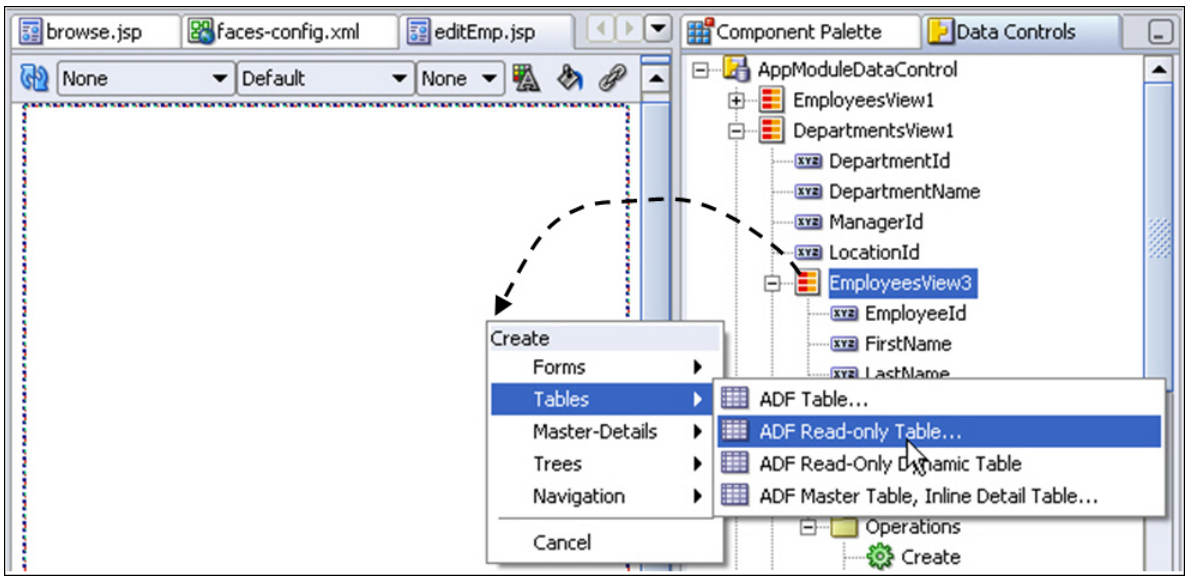


Figure 4. Dragging and dropping controls from the Data Control Palette

Application Client

As mentioned, application client code runs in a Java Virtual Machine on the client. It does not require an application server because all application code is stored and runs on the client machine although it may access business services code such as ADF BC that is deployed on a web tier server as EJB session beans. It uses libraries such as Swing (included with the Java Java SE Development Kit or JDK) to supply interface items such as windows, panels, text items, labels, and menus. Swing controls incorporate the Controller layer so no additional framework is needed to intervene between the View and Model layers of an application client module.

Web Client

ADF supports a number of web client view technologies that display in a browser. For web client applications, the code is run on an application server and generates a page that is displayed in the client's display device (such as a web browser). These technologies are primarily coded with tag languages that are supported by various Java class libraries (called *tag libraries*). These libraries present the user interface using standard controls for the appropriate device (such as HTML in a web browser). When the user submits a web page that contains input values, the page is processed by the Controller layer.

Although you can use any View layer technology in JDeveloper, the following technologies are fully supported by the JDeveloper tools:

- **JSP technology** JSP is a popular Java EE coding style that combines HTML and JSP tags. You can also embed Java snippets that perform programming functions such as conditional processing and iteration. JSP files run on an application server and usually output HTML to the client browser.
- **JSF framework** JSF is an evolution of JSP technology. It includes controller services and a full life cycle that processes page submissions, drawing interface components, and updating the Model values. This is the style that the industry is turning to for new applications because it has more features and is now a standard in the Java platform specifications. JDeveloper 10.1.3 and later fully support JSF work.
- **ADF UIX** ADF UIX is an Oracle framework that defines a page using XML code. A unique feature of UIX is its rich container model, which allows you to easily create a standard look-and-feel for your application. Since the page definition is XML metadata, the page can be rendered using different viewers such as mobile devices or a desktop browser. Oracle's E-Business Suite (Oracle Applications) currently uses UIX technology for its self-service applications although the new Fusion Applications are being written using JSF. UIX has many features in common with JSF technology. JDeveloper version 10.1.2 and earlier supports work with ADF UIX. JDeveloper 10.1.3 does not support ADF UIX.

JSF AND ADF FACES

As mentioned, Oracle is using a specific technology stack to create Fusion Applications: ADF BC and ADF Faces. ADF Faces is a set of libraries that follows the JavaServer Faces standards so the story of ADF Faces starts with a brief introduction to JSF.

WHAT IS JSF?

JSF evolved from the need to make JavaServer Pages development easier and more reusable. JSP code contains a mixture of standard HTML markup tags and JSP-specific tags that can include embedded Java snippets (scriptlets) for processing logic and references to Java class files (action tags) for performing operations (such as database queries) and generating additional HTML tags. JSP developers found themselves creating reusable libraries to perform high-level operations such as displaying the results of a query in a multi-row, multi-column HTML table. This required the use of frameworks not included in the Java standards or customized libraries, which were even more non-standard.

In addition, JSP technology is only a solution for the View layer. It requires a separate framework for Controller functions. The most popular framework used as a JSP Controller layer, is Struts, but this framework is not supported by the Java standards so integration with JSP (which is a Java standard) is the responsibility of the developer.

JSF evolved from the need for standards for this type of high-level component (for example, the table for query results) and for a built-in Controller framework. JSF is included in the Java EE 5 specification and is therefore a recognized standard.

In principle, JSF technology supports any type of client device and coding style. However, as of this writing, Sun Microsystems offers a *Reference Implementation (RI)*—code libraries that prove that the standard supports real code—only for servlets (as class libraries) and JSP technology (as tag libraries). These libraries contain classes and tags for the components and functionality described later in this section. Since RI libraries are tested and proven implementations of the standard, you can use them as a basis for your own code. The libraries included with the RI follow:

- **JSF Core** This tag library contains components, such as validators and converters, that are used in conjunction with other components. Components in this library use the "f" prefix, for example, `f:loadbundle`.
- **JSF HTML** This tag library contains HTML user interface components, such as text input, buttons, labels, and radio options. You refer to these components using the "h" prefix, for example, `h:datatable`, `h:column`, and `h:form`.

The RI also includes a *render kit* (a code layer that writes a particular kind of output format) for HTML output from JSP components although the JSF standard supports development of render kits for other client devices such as PDAs.

JSF RUNTIME AND JSF CODE

JSF code runs on an application server and the JSF runtime sends a markup language (such as HTML) to a client device (such as a web browser). A servlet (called the *JSF Servlet*) on the application server parses and interprets the code at runtime.

The files you are responsible for creating in JSF work (for example, for a web browser client) follow:

- **JSP page file** The user interface code is contained in a JSP page (with a .jsp extension) or JSP document (with a .jspx extension). This file includes all layout elements such as text fields, buttons, and graphics that the user will see and interact with. XML tags are used to draw these interface items. At runtime, the JSF Servlet parses the XML and runs a Java class file from a tag library for each tag. Property values coded into the interface component are passed to the Java class as parameter values; the Java class then performs all actions needed for that component.
- **Backing bean file** You can supplement the standard behavior of the JSF code by creating a *backing bean*—a custom Java class file for each page file. The user interface components are available as setter and getter methods in this file, so you can modify or supplement the normal behavior of the component. For example, if you wanted to modify the standard behavior of a button component, you would write a method in the backing bean. This type of code is like trigger code you would write to specify the behavior of or validate data in an Oracle Forms user interface item. The backing bean code is doubly similar to trigger code because it is usually a snippet; you are only responsible for the customized behavior, not all the additional code required to perform the normal function of the component.
- **faces-config.xml** This file (technically called the *application configuration resource file*) defines how the JSF Controller will operate. As the file extension suggests, it is XML code that defines, among other functions, the page flow (how one page calls another).

ADF FACES

ADF Faces is a set of JSF components that adds to JSP pages with JSF components a set of rich user interface controls. ADF Faces is highly integrated with JDeveloper, and the developer's experience is closer to the experience in Oracle Forms than anything else that has come before. In addition, as mentioned earlier, ADF Faces is one of the technologies being used to build the Fusion Applications. It is an evolution of ADF UIX but is written to comply with the JSF standard.

> **Note**
>
> Oracle donated ADF Faces to the open-source Jakarta MyFaces Project (myfaces.apache.org) in early 2006. This places its source code in the public domain, which means that, although Oracle will enhance and support ADF Faces for its customers, the larger Java community will also contribute to its functionality. The ADF Faces part of the MyFaces project is called "Trinidad."

ADF Faces improves on the JSF reference implementation libraries in the following areas:

- **A larger component set** ADF Faces provides over 100 tags, many of which are visual user interface items such as container components for various styles of layout, menus, trees, tables, shuttle controls, button bars, and selection lists. In addition, ADF Faces also includes date and color pickers as well as media viewers. The upcoming section "ADF Faces Components" describes some of the available components.
- **More layout components** ADF Faces ships with a set of tags that can be used as layout containers for components. For example, the `af:panelPage` tag draws an HTML table with a set of prebuilt areas for various components, such as branding graphics, a tab header, global navigation buttons, a content area, and a copyright area. Including this tag on a JSF page allows you to plug components into those areas without having to worry about how the HTML table cells and rows are drawn to maintain the arrangement of those components.
- **More properties** ADF Faces provides properties for features such as *Hint* and *Label* on an `af:inputText` item. Without using ADF Faces, you would need to add RI output text items for both the hint and the label.
- **Partial page rendering** Several ADF Faces components, such as `af:tree`, `af:treeTable`, `af:menuTree`, and `af:showDetail`, offer *partial page rendering* (PPR)—a combination of prebuilt JavaScript and HTML frames that allow just a section of the page to be redrawn. A good example of a PPR component is `af:table`. (JSF tags are best identified using a prefix—in this case "af"—that identifies the library where the tag is located in addition to the tag name.) The `af:table` component draws a standard HTML table, but adds column headings that allow the user to click and automatically sort the rows by the values in the column. When sorting in this way or scrolling through sets of records, only the table data area is redrawn. The rest of the page remains static. This results in a more interactive interface that can enhance user productivity. PPR uses the same core technologies as *Asynchronous JavaScript and XML (AJAX)*, which currently has much Java industry attention because of its enhanced user experience
- **Automatic graphics file generation** Some components, such as `af:commandButton,` create graphic files when run. This eliminates the need to create and manage separate files for graphics. The *text* property of the tag specifies the label that will appear on the graphics file. For international support, you can bind the *text* property to a message bundle file that is specific to the language of the user. That way, the label will be dynamically loaded based on the user's language preference. An example of button code and how it is displayed appears here:

```
<af:commandButton actionListener="#{bindings.Commit.execute}"
    text="Save"
    disabled="#{!bindings.Commit.enabled}"/>
```

- **ADF BC support** As with any standard Java EE View layer technology, ADF Faces work well with ADF Business Components. You can quickly bind values from ADF BC to ADF Faces component properties. Also, the Data Control Palette in JDeveloper provides automatic binding of some components, such as master-details layouts.

ADF FACES COMPONENTS

Figure 5 shows some simple ADF Faces components from the Core library that are focused on displaying single-valued items. Figure 6 shows some more complex ADF Faces Core components that are used for layout, navigation, and displaying of multiple rows or values. You will recognize some of these components because they have close parallels in Oracle Forms. Components available include the following:

- **af:selectInputDate** This component displays a text item with a graphical LOV button. When the user clicks this button, a calendar window will appear and wait for the user to scroll through the months and select a date. The selected date will be returned to the associated text item.
- **af:inputText** This component is a standard text item that includes a prompt property (like the Prompt property of a Forms item). Another of its many properties is *Required* (like the *Required* property of a Forms item); 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 prompt to suggest to the user that this field value is mandatory.
- **af:selectOrderShuttle** This component (shown in Figure 6) displays a shuttle control—two text areas filled with selections—that allows the user to select more than one value for a single field. All headings, graphics buttons, and text areas are built into this component and its child components.
- **af:commandMenuItem** This component displays a button (action item) formatted as a tab, navigation bar item, or subtab determined by which container component surrounds it. When the user clicks this item, an event is triggered.

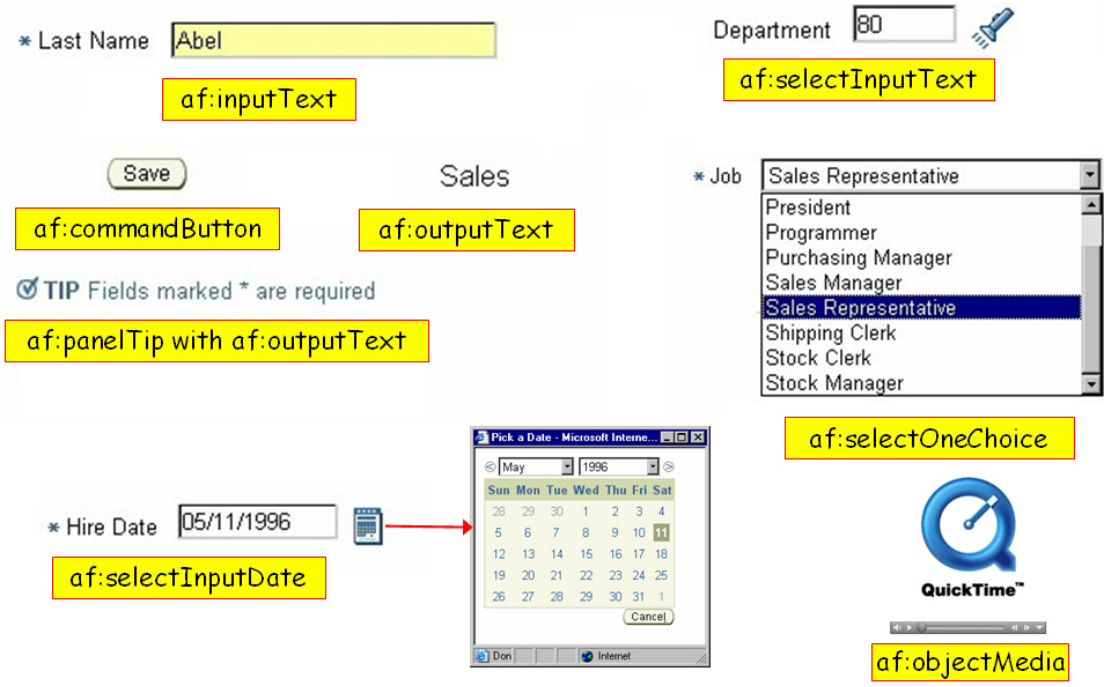


Figure 5. Item-oriented ADF Faces components

When working with ADF Faces, you do not code HTML tags directly. Instead, you use ADF Faces components that will be rendered in a web browser using HTML tags. For example, ADF Faces offers a component, `af:inputText`, which renders an HTML form input item when displayed in a web browser. The `af:table` and `af:column` components render as an HTML table.

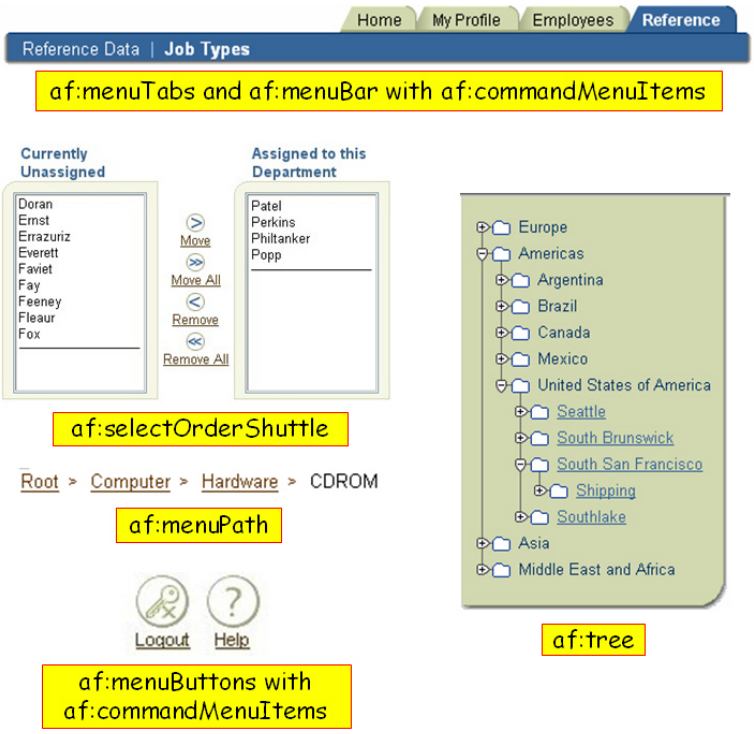
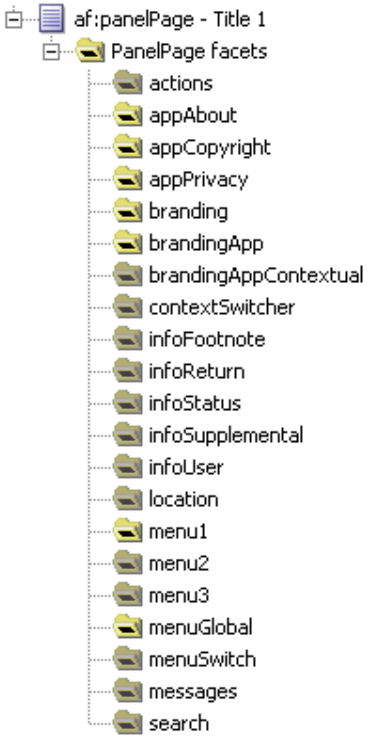


Figure 6. Other ADF Faces components

Container Components



Some ADF Faces tags act as containers, which hold other components. The process of creating a file with ADF Faces usually starts with adding container component tags, and then inserting other UI components within the container components. ADF Faces offers containers, such as `af:panelPage`, that contain a number of *facets* (predefined areas), as shown in the structure window navigator on the right.

These predefined areas use JSF facets to provide default positions for objects such as navigation buttons, copyright information, and branding logos. After dragging this component onto the page, you can drag other components into each of the facet areas you need to use.

ADF Faces containers have a parallel concept in Oracle Forms, where, before adding user interaction objects such as items and checkboxes, you create windows to contain canvases that, in turn, hold the user interaction objects. All ADF Faces containers provide automatic layout capabilities that reposition child components when the container is resized. This capability is somewhat like that of frames in Oracle Forms, although in the case of ADF Faces, the container's automatic layout applies at runtime as well as at design time.

Tag Libraries

Just as JSF offers Core and HTML tag libraries, ADF Faces offers Core and HTML tag libraries (prefixed with "af" and "afh," respectively). Some of the ADF Faces components are parallel to JSF RI components. For example, the JSF RI component `h:inputText` has an ADF Faces equivalent, `af:inputText`, used to present a text entry field. You can distinguish between these components in code, because of their prefixes ("h" for JSF RI and "af" for ADF Faces).

ADF Faces Properties

In general, ADF Faces components offer more properties and built-in functionality than the JSF RI components. For example, the ADF Faces `af:inputText` component includes a label property (for a prompt) that is not included with the comparable JSF RI component. This property is much like the *Prompt* property of an Oracle Forms item because it is a

property of the object, not a separate boilerplate object. Also, unlike the JSF RI component, the ADF Faces component validates items marked as required using JavaScript, which does not require a communications trip to the server.

Figure 7 shows a property list for an ADF Faces component, `af:inputText` (that corresponds to an Oracle Forms text item with *Item Type* of "Text Item." If you are an Oracle Forms developer, you will see many familiar properties (cross referenced in Table 1) just as you will find other familiar features as you continue to work with ADF Faces and JSF.

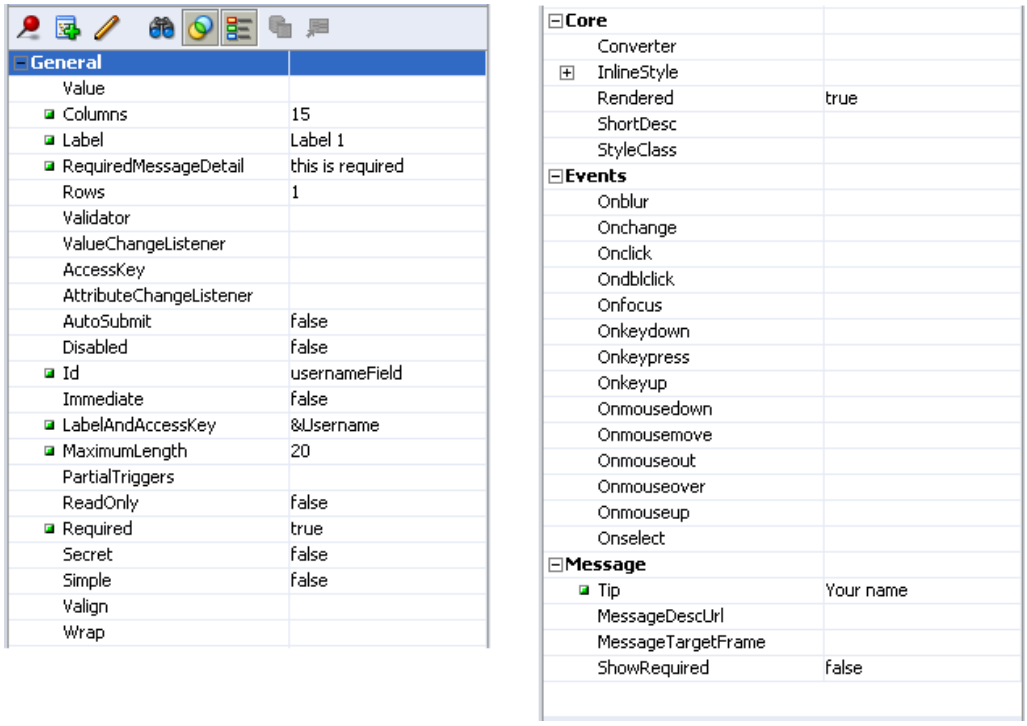


Figure 7: Property list for an af:inputText component

ADF Faces af:inputText Property	Oracle Forms Item Property
Access Key	Access Key (only for a push button or radio group item)
Binding	Column Name (or, if blank, the name of the item)
Columns	Width
Disabled	Enabled
Events	(Item-level triggers)
InlineStyle	properties in the Color and Font property categories
JavaScript Events	(item-level triggers, not properties)
Label or LabelAndAccessKey	Prompt
MaximumLength	Maximum Length
ReadOnly	Insert Allowed and Update Allowed
Rendered	Visible
Required	Required

ADF Faces af:inputText Property	Oracle Forms Item Property
Rows	Height (with the Multi-Line property)
Secret	Conceal Data
StyleClass	Visual Attribute Group
Tip (appears under the item)	Tooltip (pops up over the item)
Valign (for items) or HAlign (for containers)	Justification
Value	Initial Value
Wrap	Wrap Style

Table 1: ADF Faces af:inputText Properties Compared With Oracle Forms Item Properties

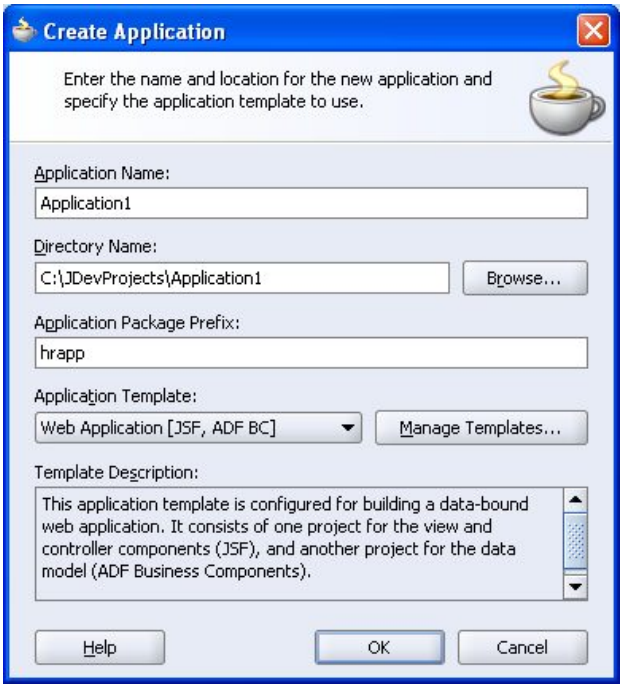
> **Note**
>
> Some Oracle Forms item properties define database-oriented functionality. This functionality is usually implemented in properties on the ADF BC objects contained in the Model layer code.

DEVELOPING AN ADF FACES APPLICATION

Although you can create an ADF Faces application in a number of different ways with JDeveloper, the process follows these general steps:

1. CREATE THE APPLICATION

As with any JDeveloper development, work starts with creating an application (application workspace) using the Create Application dialog shown here:



You can access this dialog by selecting **New** from the right-click menu on the Applications node of the JDeveloper navigator. Alternatively, you can select **File | New** from the menu to open the New Gallery, where you can select from a number of applicable objects to create.

In the Create Application dialog, you select an Application Template of "Web Application [JSF, ADF BC]," which will create two projects, Model and ViewController. These projects will be set up with the proper library references so that you can create database access ADF BC components (in the Model project) and user interface ADF Faces and JSF components (in the ViewController project).

2. Define the Model Project

Since the user interface components require database access components, you need to create a draft of the Model components next. You can use the ADF BC wizards to accomplish this (in the New Gallery's Business Tier\ADF Business Components category). Alternatively, you can use the Business Components diagrammer to draw new business components or to represent database tables and views. The diagrammer can then generate business component code.

The ADF BC objects you set up will represent the database tables and views in your application. You will set up an application module that serves as a link from ADF BC code to the View layer code. You will also set up an entity object for each table to which you need to issue INSERT, UPDATE, or DELETE statements. In addition, you will set up view objects for tables and views you need to SELECT from. To synchronize master-detail data, you set up a view link. You can refine the business components iteratively with the View layer once you have the basic components defined.

3. Create the JSF Navigation Diagram

Once you have a rough cut of the business components created, you can start developing the View layer user interface code. You can start with the JSF Navigation Diagram (shown in Figure 8). This tool allows you to lay out pages and *navigation cases* (lines that indicate which page will follow each page).

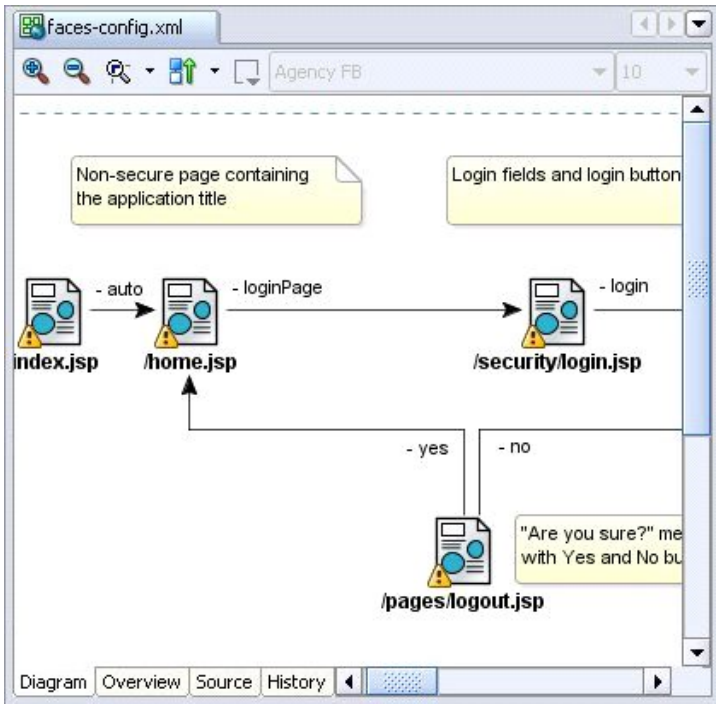


Figure 8. JSF Navigation Diagram

4. Create the JSP Pages

You can start creating the JSP pages from the navigation diagram by double clicking a page symbol to start the JSF JSP Wizard. This wizard creates a JSP file that you will use to lay out interface items. An alternative development path is to create and lay out the page files, then create the page flow diagram. The method you use is up to you, but if you start with one method (creating the JSF navigation diagram first or creating the page files first), you will probably find yourself iterating between the two methods during development. If you are creating pages before the diagram, you start the JSF JSP Wizard from the New Gallery.

5. Lay Out the Pages

The typical next step is to add the content of the pages. Using the Visual Editor, shown in Figure 9a, you drag and drop components from the Component Palette (shown in Figure 9b). You can alternatively drag and drop data controls from the Data Control Palette, shown and described previously.

The difference between dropping components from the Component Palette and dropping components from the Data Control Palette is that the Data Control Palette method creates *bound* objects—components that are connected to data or actions in the business components project. Since some components (for example, navigational controls such as tabs and non-data-oriented buttons) do not require a connection to the Model project, you will select from both palettes when laying out a page.

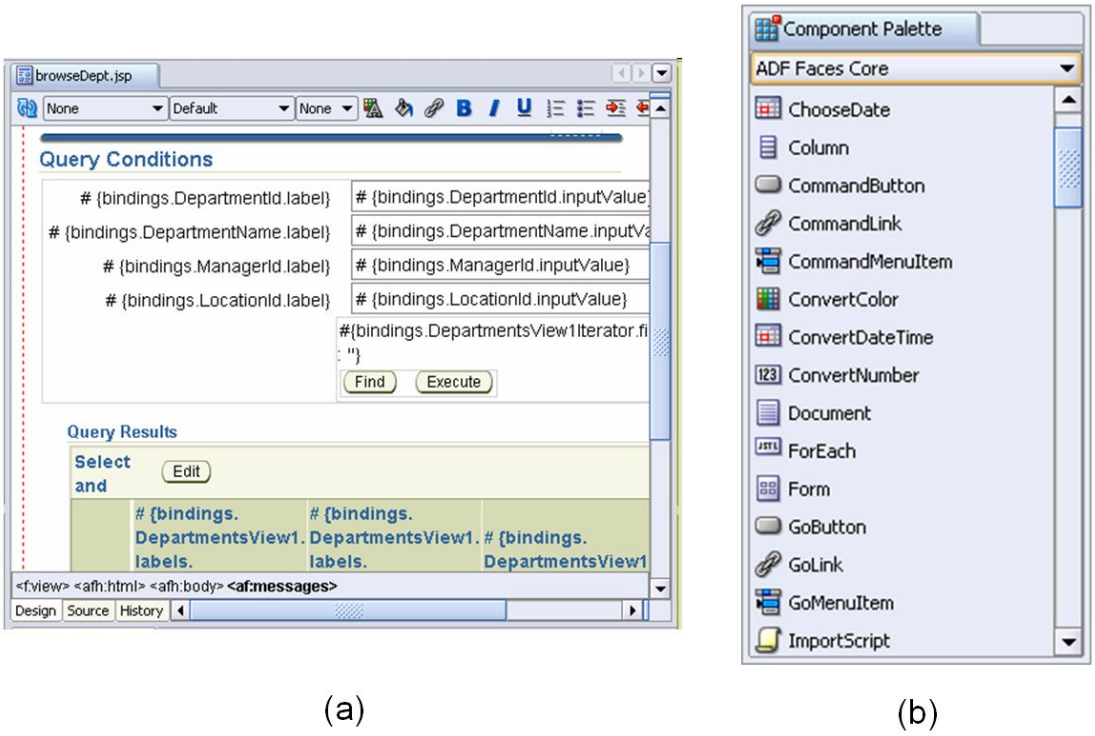


Figure 9. Visual Editor and Component Palette

The order in which you drop components and objects on the page does not matter, but it is a best practice to drop container components, such as the `af:panelPage` component first. That way, when you drop components onto the page, they can be positioned in the proper location within the containers. Selecting the proper container component is a key to effective design with ADF Faces, so you will want to become familiar with these components (look for components named with a "panel" prefix).

6. Tune the Component Properties

One of the strengths of ADF Faces is the large range of properties it offers. This feature allows you to modify the behavior of an ADF Faces component with a simple declarative setting, rather than creating a programmatic solution. The method you use for setting property values of an ADF Faces component is the same as the method you use for setting properties of an Oracle Forms control. That is, you select the component in the Visual Editor (Layout Editor in Forms Builder), and interact with the Property Inspector (Property Palette in Forms Builder). Figure 10a shows the Property Inspector view of an `af:inputText` item.

In addition to modifying components of the user interface components, you may need to modify the properties of the bindings used to connect them to the business components. You can access these bindings from the Structure window (shown Figure 10b) by displaying the binding file for the JSP page.

7. Test the Code

Running JSF code in JDeveloper is just a matter making a page active in the editor, selecting a page in the navigator, or selecting a page symbol in the JSF Navigation Diagram, and clicking the Run button. Then, JDeveloper starts the Embedded OC4J Server, which replicates the Java EE runtime environment on the Oracle Application Server. The browser will open and display the selected page. This is similar to running Forms in a standalone OC4J process from Forms Builder. Running the application before you perform modifications to the code generated by the visual environment is useful so you can check the default behavior and make note of changes, which are required.

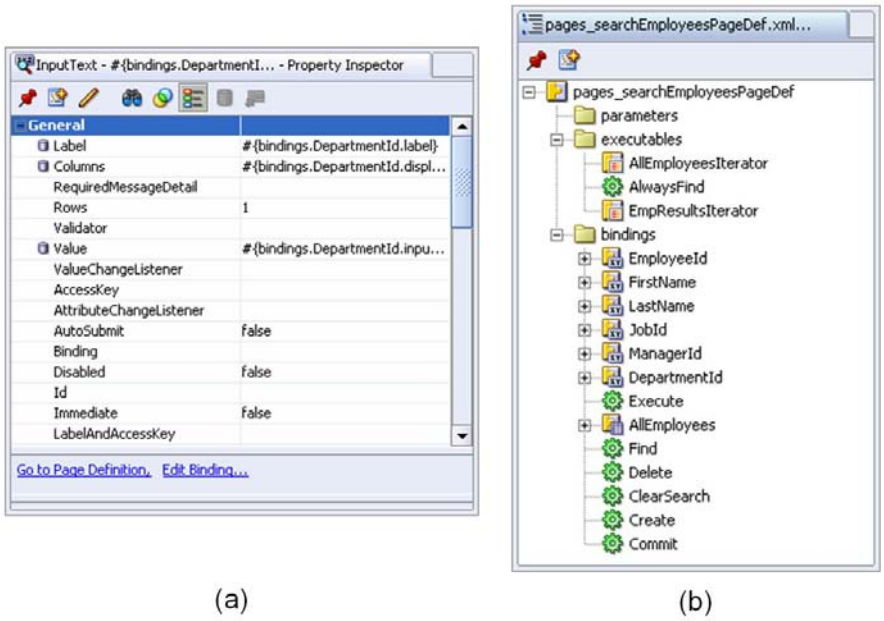
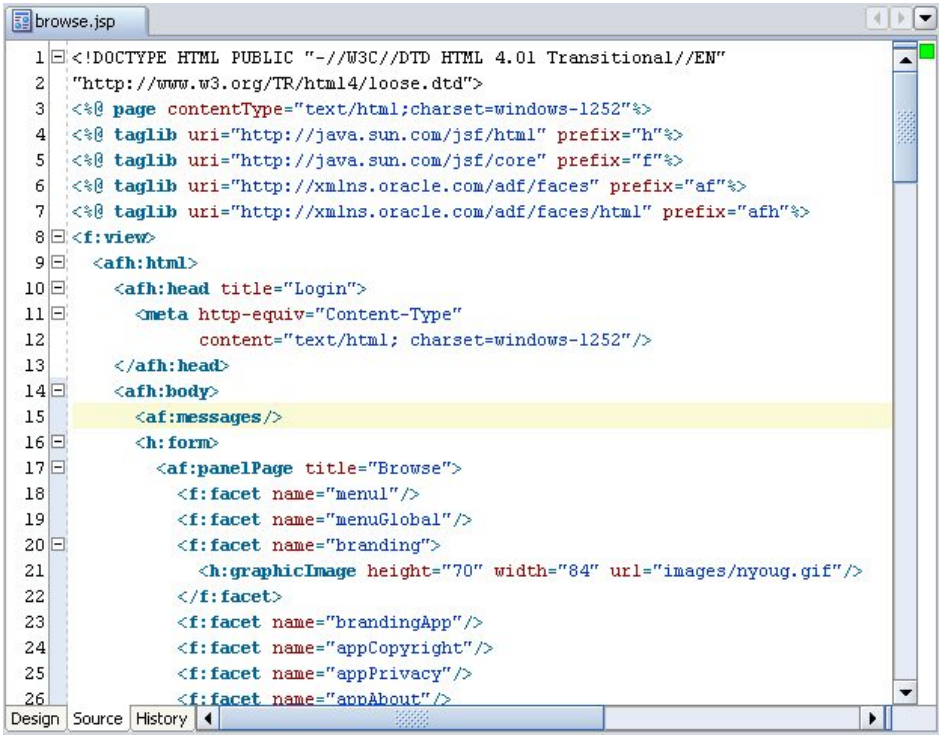


Figure 10. Property Inspector and Structure Window

8. Write Backing Bean and ADF BC Code

The preceding steps have required little or no code editing. They have used JDeveloper's visual and declarative tools to generate ADF Faces code. These methods will take you a long way towards completion of an application. However, it is most certain that they will not suffice for most enterprise application needs. At some point, you will need to write some code. All code you create in JDeveloper from any of the visual or declarative tools is available in the Code Editor (shown on the right). When you need to supplement or otherwise go beyond what is exposed in the visual and declarative environment, you will use the Code Editor.



9. Deploy the Code

After the iterative process of working with the visual and declarative tools, creating code, and testing the application is complete, you need to package the application files into an enterprise application archive (EAR) file that will be copied to the application server. All files in the Model and ViewController projects will be packaged together in the same file so that the application can be run from the application server. JDeveloper provides a deployment profile file that contains all details about the application and the libraries on which it depends. A right-click menu option in the navigator creates the EAR file based on the settings in this deployment profile.

JHEADSTART

JHeadstart is an add-on (extension) to Oracle JDeveloper that generates a large amount of UI code from declarations in an XML file. It was created and is enhanced and maintained by Oracle Consulting's Center of Excellence in the Netherlands. JHeadstart requires a separate license agreement with Oracle Consulting, but you can download from the JHeadstart Product Center's website and try a limited preview version without cost. (Search for "JHeadstart Product Center" using google.com) Once you install JHeadstart (using **Help | Check for Updates** in the JDeveloper menu), additional JHeadstart items will appear in the right-click menus and the New Gallery.

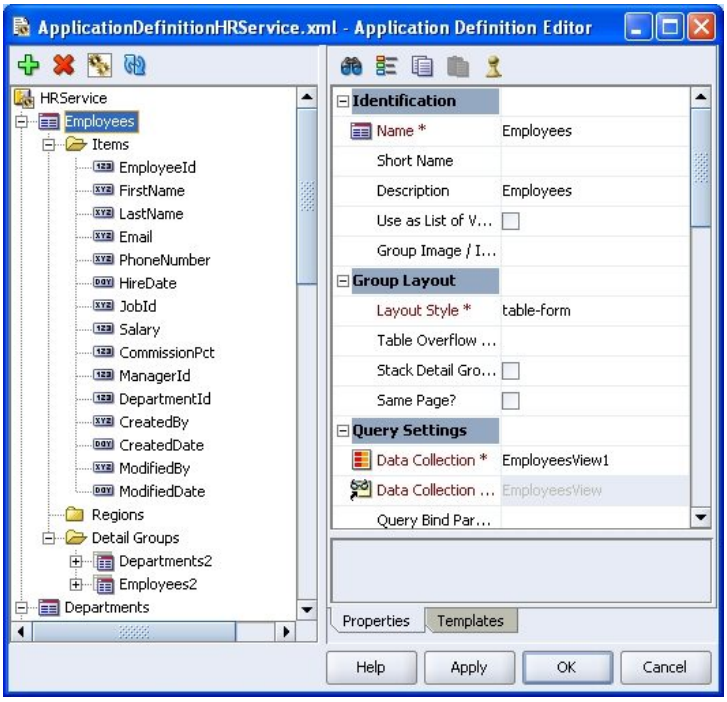
JHeadstart generates user interface code that would take much time to create manually using standard JDeveloper tools. Modifying these files uses the same techniques you use for building applications without JHeadstart. JHeadstart also generates page flow Controller code for default page styles such as form and table. It automatically adds insert, update, delete, and search functionality based on settings you make in the application definition. As an added benefit, if you are interested in using the ADF Faces tree or shuttle components, JHeadstart saves you from the significant manual effort required to properly set up those components.

> **Note**
>
> JHeadstart is not part of the Fusion Stack and is not being used to create Fusion Applications. However, it creates code using in Fusion Stack technologies (ADF Faces, JSF, and ADF BC) so all the generated code is easily maintained and integrated with any other Fusion Stack technology code.

JHeadstart offers an even more robust declarative environment than native JDeveloper and includes two generators:

- **JHeadstart Designer Generator (JDG)** This optional generator migrates module definitions you have created in Oracle Designer to application definitions for a JSF and ADF Faces application. This generator is intended to be a one-time migration tool not a bridge between environments or a way to generate Java EE code from Designer. Once you run the JDG, you work with the definitions only in JDeveloper. In addition, the JDG requires Model project definitions (ADF BC). As a precursor to running the JDG, you can run the ADF Business Components Generator to create ADF BC objects from Designer Schema Model objects.
- **JHeadstart Application Generator (JAG)** This tool creates ADF Faces JSP pages and code in the faces-config.xml file from definitions in an application definition. It is highly configurable and work in JHeadstart is oriented towards the goal of running this generator.

You will only use the JDG if you have created fully-defined modules in Oracle Designer, and you wish to move them into JDeveloper's environment without rethinking or rewriting them. In addition, after you run the JDG, you work with the migrated modules using the Application Definition Editor and you then run the JAG. Since you can use the application definition and JAG with or without starting with Designer definitions, this paper will focus on the most commonly-used JHeadstart tools: the application definition and the JAG.



APPLICATION DEFINITION

The *application definition* is a set of properties declared using elements in an XML file. The JHeadstart Application Generator uses these properties to determine the code it will generate. You interact with this file using the Application Definition Editor (shown on the right).

The editor is set up with a hierarchical navigator on one side and a property editor on the other side, much like Forms Builders' Object Navigator and Property Palette. Selecting a node in the navigator opens its properties in the property editor. The toolbar in the editor allows you to create nodes, copy and paste nodes, copy and paste properties, and generate the application. If you have used

Oracle Designer, you can think about the Application Definition Editor in the same way as the Repository Object Navigator—it exposes all elements and properties in the repository used to generate the application.

Some of the nodes in this navigator are shown in Figure 11. So that you can get a taste of the flexibility of JHeadstart and the types of code you can generate, it is useful to take a brief look at these nodes and some of the properties they offer.

Service

The JHeadstart service level applies to an entire application. It is assigned an application module from the Model project, and all view object instances declared for that application module are available to the JHeadstart nodes under the service. The service element contains properties such as the following:

- **UI Pages Directory** This property defines the top-level directory for all JSF pages generated by the JAG.
- **Overall Layout Style** This is the category of page navigation that the JAG will generate: Menu or Wizard.
- **Resource Bundle Type** The value of this property declares whether messages will be generated into a Java file or a plain property-value file.
- **Unselected Label in Dropdown List** This property declares the default null value label for pulldown items (poplists) in the entire application.

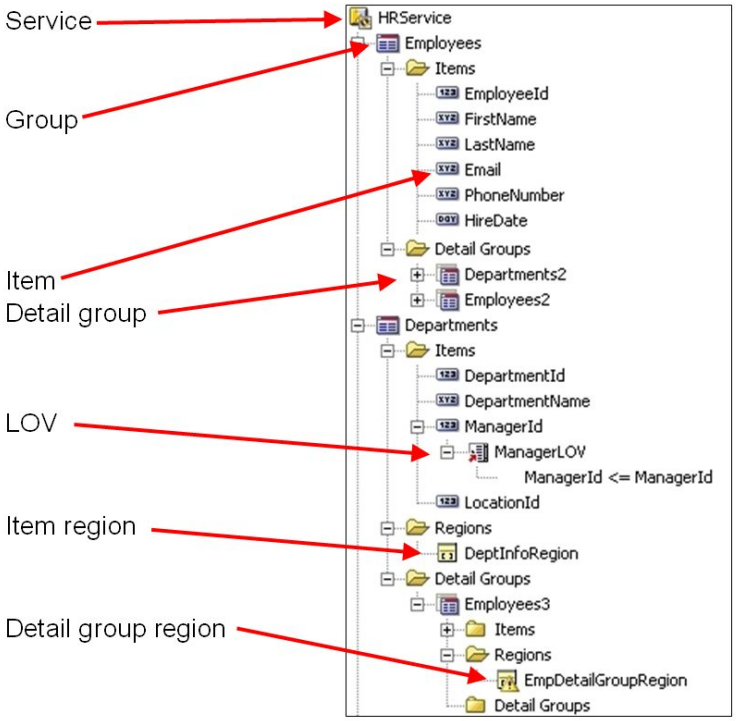


Figure 11. Application Definition Nodes

Group

The group level of properties corresponds to a single view object instance in the application module. For example, the Employees group shown in Figure 11 corresponds to a view object that represents a query to the EMPLOYEES table. Other than the view object assignment, the most important property of the group is *Layout Style,* which allows you to declare the general appearance of the page and the type of page flow that will be generated. You can assign the following values to this property to create different types of pages:

- **form** to create a single record layout page.
- **table** to create a multi-record rows and columns layout.
- **table-form** to create a page flow where the user selects a record on the table page and views it on a form page.
- **select-form** to generate a page where the user select a record from a list and views it on a form page.
- **tree** to generate a list of hierarchically-related records in a navigator style display.

- **tree-form** to create a tree page on the left and an edit page on the right containing edit fields for the node selected in the tree.
- **parent-shuttle** to create a shuttle control representing a detail table populated by the shuttle.
- **intersection-shuttle** to generate a shuttle control representing a detail table populated from two parent tables.

In addition to the basic layout and page flow style, you can define search options for each of these pages using a property of the group. The ability to insert, update, and delete records are defined as properties for the group, a well. You can think of the group as holding the same type of properties as a block in Forms.

ITEM

The item level under the group holds settings for the field- or column-level properties, just like items Forms. Some of the properties of the JHeadstart follow. You will see parallels with properties of Forms items:

- **Hint** The text in this property will display under the field on the screen.
- **Prompt** This is a field label for the item.
- **Display Type** This is set based on the datatype of the attribute. See the sidebar "A Note about Pulldowns (Poplists)" for information about setting this property for pulldown controls.
- **Column Sortable** ADF Faces table columns include a property that causes a button to be displayed in the column heading. Clicking this button will sort the rows in the table based on the value in that column. This property generates the sort behavior for this item when the item is used in table layouts.
- **Width** and **Height** The JAG generates components with default sizes. These properties allow you to define the size of the item if it should be different from the default.
- **Display** This property declares whether the item should be shown on the screen.
- **Display in Table Layout** If you set this property to "false," the JAG will not add the item to table-style layouts but will still generate it into form-style layouts (if *Display* is set to "true"). This is useful for tables with many columns. So that the user is not forced to scroll horizontally, you can exclude columns from the table display if the columns are not required to identify a record.

You have no doubt noticed that some of these properties are very similar to those explained for the `af:inputText` component earlier in this paper. This is no coincidence. In fact, the JHeadstart properties of an item in some cases (such as *Hint* and *Prompt*) will be assigned to corresponding properties for the `af:inputText` component that the JAG generates.

A Note about Pulldowns (Poplists)

You can define a pulldown list for an item with just a few definitions. First, you need to create a Domain element in the application definition. You specify whether the domain is a static list of hard-coded values or a dynamic list based on a view object. Then you set the item properties: *Domain* as the name of that new domain and property *Display Type* as "dropDownList."

DETAIL GROUP

Detail groups have the same properties as groups but they are displayed either on a separate page accessible with a Details or subtab button or on the same page as the master record. The detail group requires a master-detail view link defined in the application module as explained in the earlier section on the data model of the application module. The group's *Same Page* property defines whether the detail group appears on the same page or on a separate page accessed with a subtab or Details button click.

LOV

LOV definitions generate LOV functions for specific items. You create a separate group for each LOV view object (like a record group in Forms). The group properties are set: *Use as List of Values?* checked and *Layout Style* as "table." Then you set the item *Display Type* property to "lov" to generate a text item with an LOV button. Under the item in the navigator, you create an LOV subnode with an attribute mapping (LOV view object attribute to the attribute for the base view object on the page). This feature alone can save much time from the largely manual process that JDeveloper alone requires to create LOVs. Figure 11 shows an LOV node defined for the ManagerId item.

Note

You can assign more than one item on the screen from a single LOV group.

Regions – Item, Detail Group, and Region Container

Regions represent areas on the screen; you create regions and then assign properties and region contents to them. You can create three types of regions in JHeadstart as follows:

- **Item region** This type of region holds items. You can assign a heading for the region and define in how many columns the items will appear. This type of region is like an item group in Designer. Figure 12 shows two item regions generated into a page—one with a two-column layout and another with a three-column layout.
- **Detail group region** This region type holds detail group items. You use this for nested groups that appear on the same page as the master group. (That is, the *Same Page* property must be checked.)
- **Region container** This region type acts as a housing for item regions and detail group regions. You set a layout style of horizontal, vertical, or stacked to place objects in specific locations on the page.

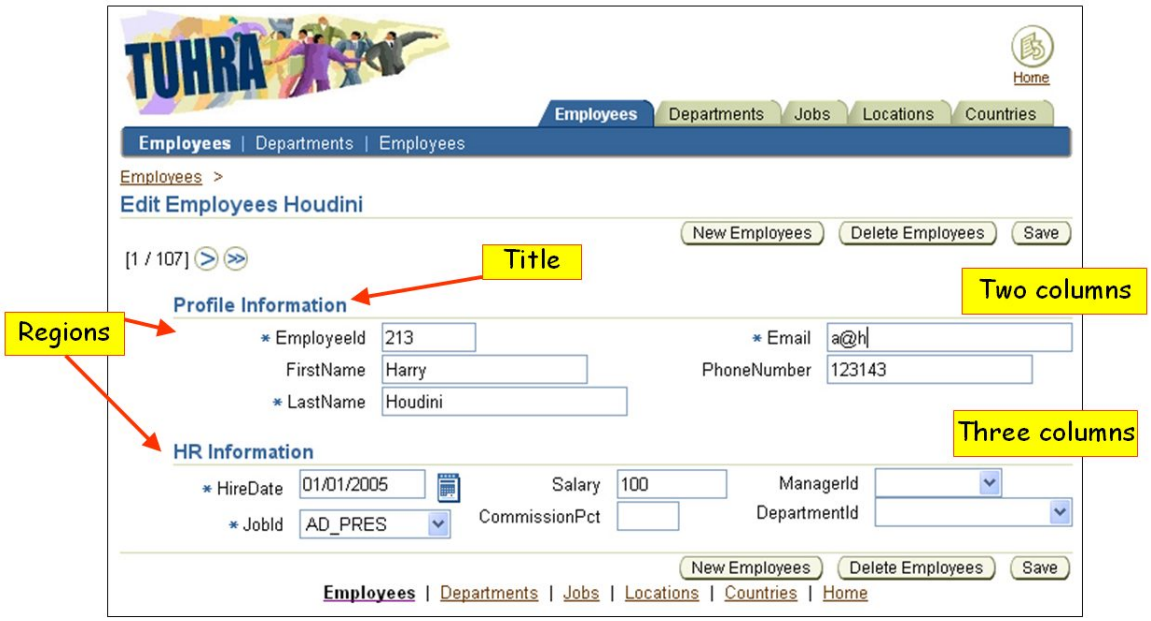


Figure 12. Generated item regions

Default Application Definition

When you create an application definition file (using the New Application Definition Wizard), you can choose to create a default definition. This default definition mirrors the application module definition's data model as shown in Figure 13. In Figure 13a, the application module data model shows top level nodes for Countries, Employees and Departments. Each top-level node has one or more sub- (detail) groups. The default application definition created from this application module (Figure 13b) mirrors this data model. Each top-level view object instance in the application module shows up as a top-level group in the application definition. Each detail view object instance becomes a detail group in the application definition. The view link instances in the data model define the master-detail relationships between the groups in the application definition.

Templates in JHeadstart

JHeadstart uses some default templates declared using the Velocity template framework (jakarta.apache.org/velocity) to generate the basic look and feel of a component, but you can replace the defaults at any level. That means if one item requires a specific template but the rest of the application does not, you can declare a specific template just for that item. You create templates most easily by copying and modifying the existing templates shipped with JHeadstart. Then you declare the name of your template on the Templates tab of the Application Definition Editor for the node to which you want to apply the template.

All elements on the page are generated using templates so you can have very detailed control of the output of the JAG. Although the Velocity templates are only used at runtime, parts of the page are included from separate common files using the

ADF Faces `af:region` tag. The `af:region` tag in the JSP file includes another file in a specific part of the screen at runtime. This means that you can define common layout elements in a small set of files. Changes to the general appearance of the application are then easier because modifying the centralized set of files will affect the look and feel of all pages.

100% GENERATION?

In Designer Forms Generator, 100% generation is always a goal because, if you have no post-generation changes, you can work only declaratively with your application and allow the generator to create the runtime code. Although this is not a guiding goal of JHeadstart (as the name implies), mastery of the templates is a key to 100% generation. Velocity provides a scripting language that can help you squeeze the most out of the JHeadstart generator. This means that you have a high level of control over the generated output (again, if you have mastered the template approach and language).

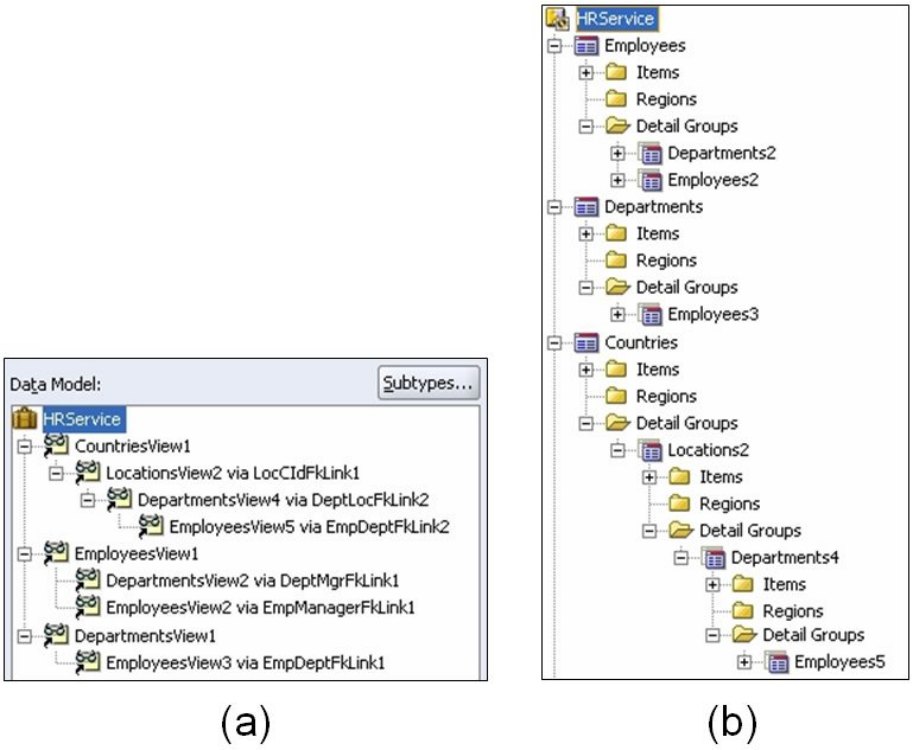


Figure 13. Application module data model and JHeadstart navigator nodes

> **Caution**
>
> While mastery of templates and the Velocity language is helpful to get the most out of the JAG, this type of work is not for novice programmers. Working with template code requires much care and experimentation. However, as with many aspects of JDeveloper work and template work in general, you can relegate the template work to senior developers and assign the less experienced developers the work of defining properties and running the generator, which will use the customized templates.

JHEADSTART DEVELOPMENT PROCESS

The JHeadstart development process consists of first creating a functional Model project with ADF BC in JDeveloper using standard JDeveloper techniques or by using the Designer business components generation tool. Then, you enable JHeadstart on an empty ViewController project. This sets up the project with the proper library references and adds templates and other files to the project directories. Next, you create an application definition file using the right-click menu in the navigator or using the New Gallery. The relationships between database objects, ADF BC objects, and JHeadstart objects is shown in Figure 14. Figure 14 also shows the JAG and its output.

After those setup steps, you iterate between modifying the application definition, generating it, and running it. This iteration continues until you have gotten as much out of JHeadstart settings as possible. You then turn off generation for one or more pages, and modify the pages manually using the standard JDeveloper tools. You can continue to tweak settings and regenerate for those pages whose generation switches are on, but eventually, you will stop generating. As with a JDeveloper application without JHeadstart, you then deploy the completed code to the application server.

JHEADSTART APPLICATION GENERATOR

The JAG uses template information and information in the application definition to create JSF pages and a faces-config.xml file. The application that you create with JHeadstart follows the "Fusion Stack" technology choices: ADF BC, JSF, and ADF Faces. You run the generator from a button in the Application Definition Editor or from a right-click menu option on the application definition file in the navigator.

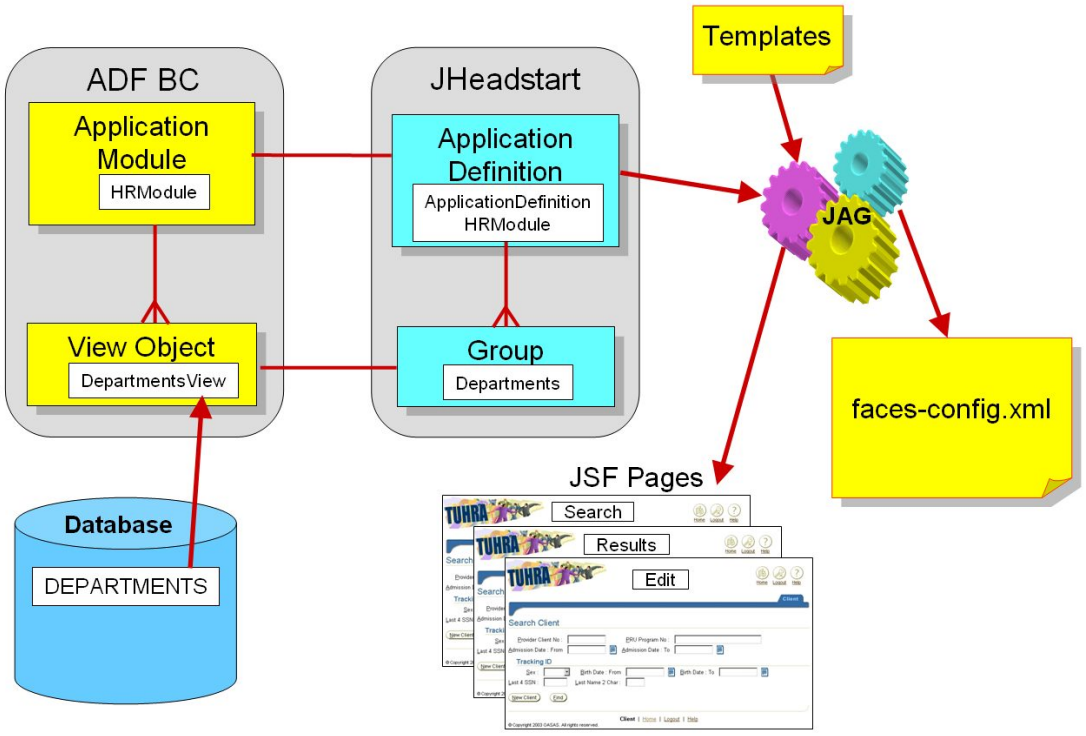


Figure 14. JHeadstart database use and code generation

The JAG reads the application definition file and builds a set of pages for each group in the definition. It reads all subgroup, LOV, and item information and generates appropriate regions, buttons, and text items on each page using the formats declared in the templates.

BEST PRACTICE

JHeadstart is updated frequently and, with the proper license, you can download patches and new releases to update your installation. The process of upgrading often includes migrating the application definition to the new release and regenerating your applications. If you have made significant changes to the generated code, you will need to redo those changes. Therefore, the JHeadstart team suggests that you keep a log of the post-generation changes you make to each page or file so that you can restore those changes after regenerating the application using the new release. This practice insulates you from version changes and allows you to take advantage of the new features and bug fixes published by the JHeadstart team.

GENERATED CODE SAMPLE

Figures 15 through 17 (generated by Steven Davelaar, Oracle Consulting, and reprinted from the book *Oracle JDeveloper 10g for Forms & PL/SQL Developers*, by Koletzke and Mills, Oracle Press, McGraw-Hill/Osborne) show samples of the type of code you can generated with JHeadstart. Each figure is labeled with the style of code generated into it. If you are accustomed to Designer Forms Generator work, you will see parallels in the types of layouts you can create only with declarations in the repository.

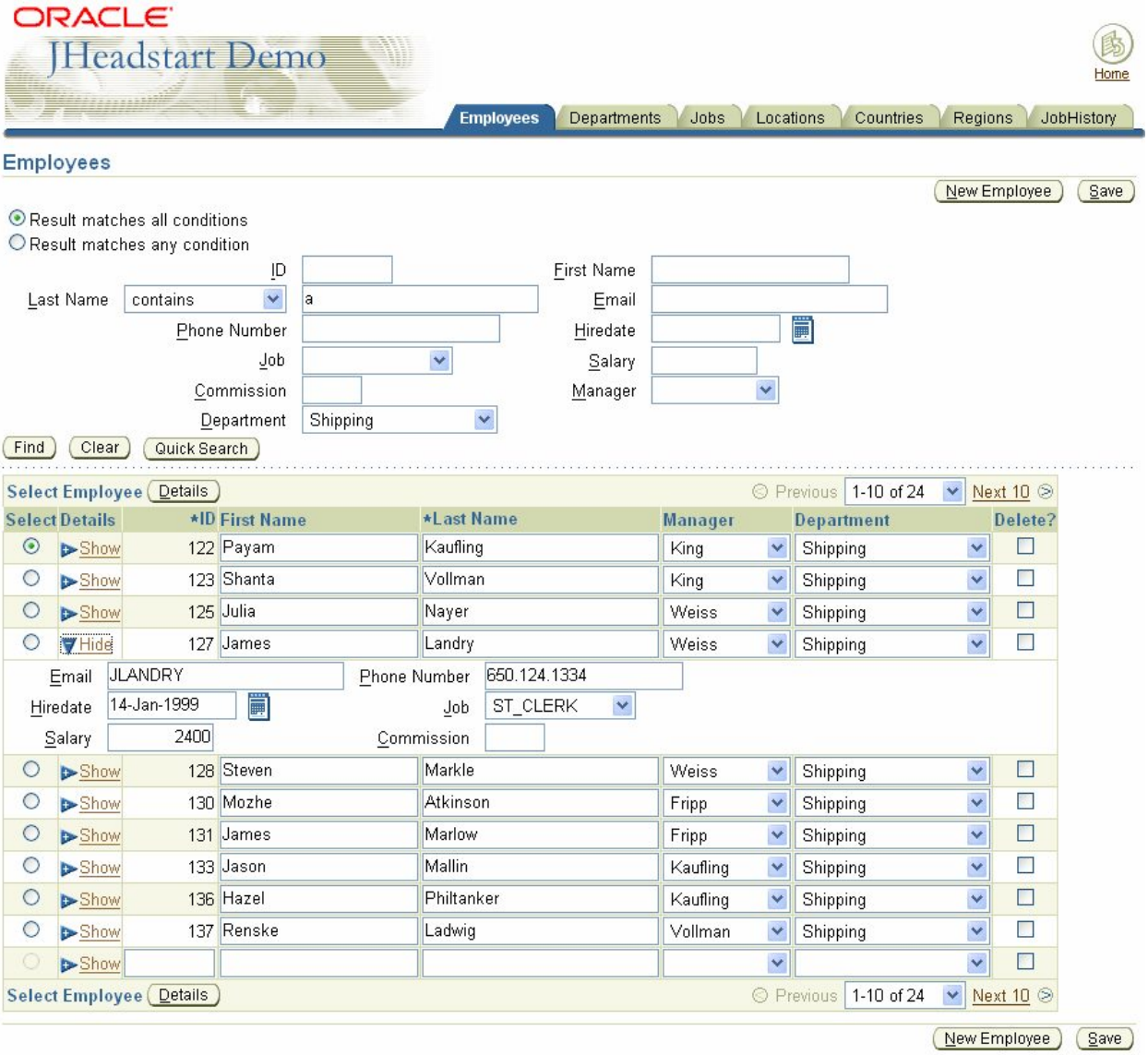


Figure 15. Advanced search, editable table, inline overflow

IS IT ORACLE FORMS YET?

Now that you've had a taste of the JDeveloper tools, ADF and the ADF development process, and the JHeadstart add-on to JDeveloper, it is time to draw some conclusions about JDeveloper's productivity and how it can help an Oracle development shop.

As JDeveloper has evolved from its first release, developing Java-oriented applications has become easier and easier. Most "traditional" Oracle PL/SQL and Forms developers have probably looked at JDeveloper sometime in the past and have rejected it as too difficult to use or as having too steep a learning curve to be productive for new applications. They have wanted a tool that is as easy as Oracle Forms for quickly creating application front end code.

RIGHT. SO, IS IT FORMS YET?

Despite the vast improvements made to JDeveloper's developer interface and the ease-of-use that ADF and its tools bring, the answer to this question is a still guarded "No, but it's a close second." To soften that statement a bit, a comparison like this is unfair. Tools oriented towards vastly different technologies can never be the exactly the same. Therefore, any new technology and new tool used to work with it will seem very foreign and require learning and adjustment. For example, consider your first introduction to Oracle Forms. It took a while for you to assimilate the fundamentals of Forms and how to think about a solution using that tool. It also took years of working with the tool to be able to address complex application requirements in a creative way.

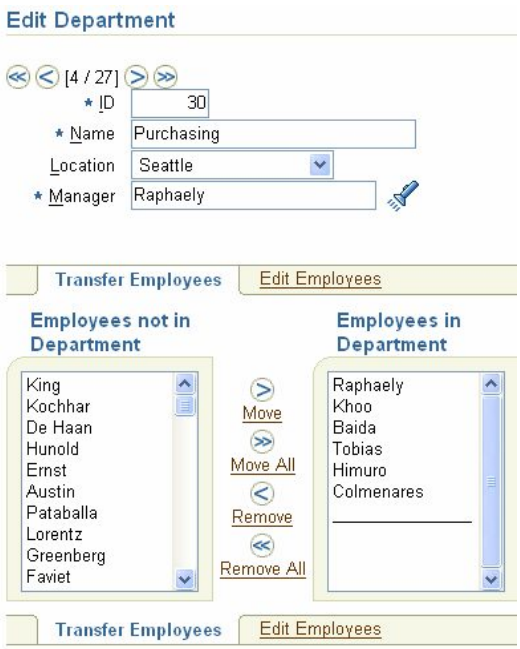


Figure 16. List of values, stacked children, shuttle

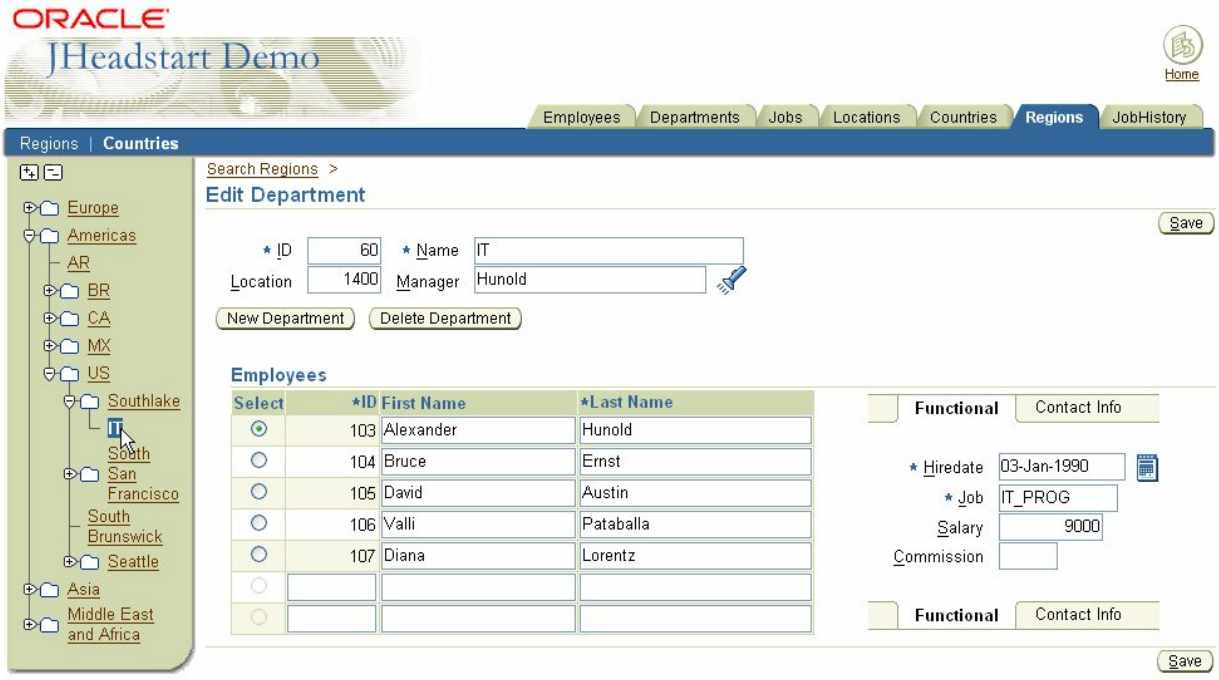


Figure 17. Tree, form, editable table, stacked overflow

There is no question that the technologies behind Oracle Forms (a proprietary 4GL environment) and JDeveloper (based on Java standards) are different and that the standard Java-based approach is seemingly more complex because it allows you to modify any code that the tools generate. Although much of the behavior of Oracle Forms is modifiable using properties, there is always some of the behavior that you cannot control without replacing items with Pluggable Java Components (PJCs), or rewriting the standard Forms DML mechanism (using ON- triggers).

Code you create in Oracle Forms is contained in a binary (FMB) source code file that can only be accessed by the Forms Builder (or API code you write in C++ or Java). On the other hand, all code you create in JDeveloper is contained in ASCII text files that you can modify with any text editor. Since everything is exposed, you have more options for modifying behavior

and this leads to a perception of more complexity. A general rule of all development environments is that the more control you have over the application, the more responsibility you have to write code that makes it work properly. JDeveloper softens this responsibility greatly by generating a large amount of code needed for *plumbing (*sometimes called *scaffolding)*—standard internal operations that include communication between various layers of the application.

A QUICK COMPARISON

The technologies are different. It naturally follows that the development methods are different so a comparison at those levels is not fair. It is possible to offer some comparisons between the tools in the realm of productivity of various development operations as well as in the realm of general features. Table 2 attempts such a comparison. Some of these statements contain disclaimers that they are "apples and oranges" comparisons because the environments are so different.

Development Operation or Feature	The "Winner"
Declarative and visual environment	**A tie**: Both have the same basic capabilities. Forms' Layout Editor allows you to place components precisely, but JDeveloper's Visual Editor allows you to add container components that manage the layout and maintain relative positions even when the window is resized at runtime. JHeadstart development is completely declarative although you may need to modify some visual aspects as a post-generation change. In its current version, JHeadstart does not offer visual editing of the Velocity templates from which all objects are generated. This is not a severe limitation because you can verify the layout of an object in a standard JSP file using the JDeveloper visual editor. You then copy and paste the resulting code into a Velocity template.
RAD for standard functions	**A tie**: Quick prototyping of standard DML and query operations require about the same amount of time and effort in both tools. As a related sidebar, JDeveloper has won several industry competitions over other Java IDEs for development speed and accuracy.
RAD for complex functions	**Neck and neck**: Forms is slightly ahead of JDeveloper without JHeadstart. With JHeadstart, some complex functions (such as advanced search features, LOVs and table-based poplists) are automatically generated so JDeveloper with JHeadstart is ahead of Forms for some UI requirements.
Complex development tasks	**Also neck and neck**: The considerations here are similar to those of the RAD functions. With an even playing field of developers who are in tune with the way each tool works, if the complexity of the need is great, it will require creative solutions in both tools. However, both have strengths that the other does not. For example, creating table-driven poplists in Forms requires three elements and at least three lines of code. In JDeveloper, the definition of the query in a view object and a few quick selections in a binding editor accomplish the same task in a faster way. LOVs, on the other hand, are more time consuming to create in native JDeveloper 10*g* than they are in Forms, although the time to create an LOV in JDeveloper with JHeadstart is much less than the time required in Forms. (JDeveloper 11g will offer more native LOV support, which will may allow JDeveloper to inch ahead of Forms in this category.)
Learning curve time	**A photo-finish, but Forms will show as the winner in the photo**: Forms hides a large amount of internal plumbing, whereas JDeveloper allows you to access it if you want to. If you need to modify the internals, you need to fully understand the Java EE web application environment. You can still be productive with minimal knowledge of this environment, but you will not be able to have ultimate control over the runtime without much study. Since Forms does not allow you to access the internal runtime level, you do not have the option of modifying it (and, therefore, do not need to worry about it).
Ease of extensions	**JDeveloper is slightly ahead**: This is related to the preceding item. Extending Forms requires writing PJCs or calling external Java class files. Extending JDeveloper code is a natural and expected part of development work. JDeveloper allows you to access all code that it creates and any code that you add will be revealed in the appropriate code editor or property palette. For example, extending class files is a natural task when working with Java,

Development Operation or Feature	The "Winner"
	so adding your own functionality to an ADF BC entity object is native to that framework. An interesting note: the tool of choice for writing Forms PJCs is JDeveloper.
Interactivity of user interface components	**Forms wins by a length but a new race will be needed in early 2008**: This is not a fault of the tool; it is an effect of the runtime environment. Forms (from release 9i on) can only run code within an applet session of a web browser. Applets are run in a client-side JVM; they use highly-interactive user interface controls from libraries such as Swing (included with the Java JDK). JDeveloper web applications, like all web applications, are run on an application server and only display HTML-oriented controls such as HTML items, buttons, and forms in a web browser, which is only knowledgeable about HTML. Adding JavaScript to the HTML controls can make them more interactive, and ADF Faces builds JavaScript into the partial page rendering feature discussed earlier. In addition, an upcoming release of ADF Faces will include richer client components that take PPR beyond its current level. Therefore, the future is bright for emulating the high interactivity of Forms applications using HTML web browser components, but for now because of its nature as a client-side runtime tool, Forms is ahead.
Number of user interface components	**JDeveloper by ten lengths**: ADF Faces offers over 100 components (a handful of which are not visual) as opposed to 22 components in the Forms palette (8 of which are drawing shapes).
Future enhancements and support	**JDeveloper although support for Forms will not vanish**: Oracle customers are heavily invested in the Oracle Applications and in custom-built Forms applications. Oracle has stated that there is no foreseeable end to support for Oracle Forms, but Oracle Forms has been "functionally stable" for a number of years; enhancements to Oracle's development tools will be made primarily in JDeveloper. JDeveloper is also a supported tool and, since it is being used to create the Fusion Applications, its support also has no foreseeable end.
Popularity	**Forms but only because it is more mature**: Based on quick surveys done at conference presentations, it seems that many traditional Oracle shops have noticed Oracle's focus in the Java EE world over the past several years and are starting to build new applications using Java EE technologies. JDeveloper has been waging an uphill battle in the Java development world against popular open source tools such as Eclipse. Even though JDeveloper exceeds the native capabilities of other Java IDEs, Java developers are wary of a non-open source tool. Traditional Oracle shops, however, usually select JDeveloper as the tool because they prefer a single-vendor solution for their software. They also prefer a vendor-supported tool over an open source, community-supported tool. In addition, they understand that Oracle is striving to make JDeveloper easier and easier for Forms developers. This cannot be said of any other Java EE development tool on the market. Moreover, Oracle is proving its interest in supporting JDeveloper and ADF for the long term by building the Fusion Applications using those tools. This is the primary reason most organizations chose Forms for their own custom applications—that Oracle Applications was built with it, and it is the same now for JDeveloper and ADF Faces.

Table 2. Comparison of JDeveloper and Oracle Forms Development Operations and Features

Note

This paper does not attempt to address the decisions needed when migrating existing Forms applications to a JDeveloper environment. Although many third-party solutions exist for this type of migration, from experience, if you are faced with a migration, it is best to not only seriously evaluate these migration products, but also seriously consider scheduling and budgeting for a rewrite of the application. Rewrites are usually to be avoided regardless of the technologies, but in this situation, a rewrite might serve you better from the standpoint of maintainability in the long term.

CONCLUSION

If you are a traditional Oracle shop, JDeveloper with the Fusion Stack and, optionally, JHeadstart offers an environment in which you are likely to be productive. As with any new environment, it is best to focus your first development effort on a small application, if possible. That way, you will be able to write and rewrite several times with less overall time impact. Your first application will give you experience and (hopefully) the satisfaction that this toolset can eventually make you as productive as you are in Forms.

In addition to adjusting to the ADF method of development, you will also need to know Java (at least at a novice level) so that you can write small snippets of code to extend the framework functions. Since JDeveloper hides much of the internal plumbing for a Java EE web application, developers can be very productive without knowledge of everything in the Java EE world. Although you can be productive as a development team with much of the team knowing Java at the scripting level, you also need a Java developer who has architect-level skills. This person will guide the Java development of the rest of the team and will write the framework extensions that are occasionally needed.

Developers will also need to have a basic grasp of HTML and XML, but fluency is not required with these either. You will actually code very little, if any, HTML because ADF Faces generates HTML for you. You will also code very little XML because JDeveloper's tools and wizards do that for you. The main fact you need to know about HTML and XML is that tags consist of the element (main name of the tag) and attributes (or properties) of that tag, which you use to modify or define the element's behavior. You also need to understand that some tags (such as `f:facet`) only make sense between the start and end tags of another tag (such as `af:panelPage`). Therefore, the arrangement of both HTML and XML tags within a file is hierarchical (tags within other tags).

This paper has described JDeveloper, ADF, the Fusion Stack, and JHeadstart tools and offered some opinions about what you might think about when selecting a tool for a new application. Hopefully, this will help you when you need to make a decision about using JDeveloper and the Fusion Stack for future application development needs.

###

Peter Koletzke is a technical director and principal instructor for the Enterprise e-Commerce Solutions practice at Quovera, in Mountain View, California, and has 25 years of industry experience. Peter has presented at various Oracle users group conferences more than 220 times and has won awards such as Pinnacle Publishing's Technical Achievement, Oracle Development Tools Users Group (ODTUG) Editor's Choice, ECO/SEOUC Oracle Designer Award, ODTUG Volunteer of the Year, and NYOUG Editor's Choice. He is an Oracle Certified Master, Oracle ACE Director, and coauthor of the Oracle Press Books: *Oracle JDeveloper 10g for Forms & PL/SQL Developers* (with Duncan Mills); *Oracle JDeveloper 10g Handbook* and *Oracle9i JDeveloper Handbook* (with Dr. Paul Dorsey and Avrom Roy-Faderman); *Oracle JDeveloper 3 Handbook*, *Oracle Developer Advanced Forms and Reports*, *Oracle Designer Handbook*, *2nd Edition*, and *Oracle Designer/2000 Handbook* (all with Dr. Paul Dorsey).