Oracle JHeadstart Overview

An Oracle Technical White Paper
June 2006

ORACLE

Oracle JHeadstart Overview

Oracle JHeadstart Overview

The productivity rates achieved in developing java-based web applications are often disappointing. If you have a background in 4GL development tools like Oracle Designer and Oracle Forms, you might have experienced that functionality that used to take a few hours to build, now easily takes days. If you are a Java web developer without a 4GL background, you might be confronted with more and more demanding (internal) customers who press you to deliver a maximum of functionality in the shortest possible timeframe.

What you need in such a situation is a development environment that is based on opens standards, and delivers easy to maintain applications quickly. Oracle offers such an environment with Oracle JDeveloper, the Oracle Application Development Framework (ADF) and Oracle JHeadstart.

Oracle JHeadstart is an extension to Oracle JDeveloper that strongly increases developer productivity in a J2EE environment. By specifying the application metadata, using a simple property editor, and using a powerful generator that assembles reusable user interface components to implement recurring user interface patterns, you will be able to develop complete and feature-rich transactional web applications very rapidly. The new version of Oracle JHeadstart is fully based on the open standard JavaServer Faces.

This white paper provides an overview of Oracle JHeadstart. It explains what it is and describes what you need to do to develop an application from scratch using JHeadstart. In the last section it addresses how you can use JHeadstart to generate applications from Oracle Designer and migrate Forms applications to J2EE.

This is a technical whitepaper and is targeted at Java developers who want to learn how they can be more productive, Oracle Designer/Forms developers who want to make the move to J2EE, and system development managers who want to learn more about JHeadstart to implement it in their development environment. It can best be read when you have some knowledge of Oracle ADF. For more information please refer to the resource list in the last section of this white paper. Although references are made to Oracle Forms and Oracle Designer it is not necessary to have any knowledge of these products.

This white paper is a largely based on Chapter 16 of "Oracle JDeveloper 10g for Forms & PL/SQL Developers: A Guide to Web Development with Oracle ADF"

(McGraw-Hill/Osborne, Oracle Press 2006) by Peter Koletzke and Duncan Mills, ISBN 0072259604. The book is not yet released, but can be preordered on Amazon.com.

WHAT IS JHEADSTART?

Oracle JHeadstart represents worldwide best practices of Oracle Consulting collected during development of large transactional web applications.

JHeadstart is a product that is developed and maintained by Oracle Consulting; it is a development toolkit built on top of ADF that uses JDeveloper's extension API to fully integrate with the JDeveloper IDE. JHeadstart generates fully functional, ADF-based web applications with features such as wizards, trees, shuttles (multi-select fields), LOV's with validation, advanced search, quick search, and role-based security. JHeadstart generates these from simple metadata properties.

JHeadstart Benefits

JHeadstart can deliver the following benefits:

- **Higher productivity** It offers a significant productivity advantage in creating sophisticated web-based, J2EE, ADF business applications.
- **Easier transition to J2EE** It will ease the transition to J2EE and ADF as it builds on concepts familiar to people with an Oracle Designer and/or Oracle Forms background. You use wizards and property editors to declaratively specify advanced behaviors that are generated into your application.
- **Consistent look and feel** Using an application generator such as JHeadstart results in a highly consistent look and feel for your application.
- **Ability to leverage of Oracle Designer work** JHeadstart protects your investment in Oracle Designer since it can migrate Oracle Designer metadata to build ADF applications.
- **Ability to leverage Oracle Forms work** JHeadstart can be used to migrate Oracle Forms applications, either generated using Oracle Designer or hand-built using Oracle Forms.
- **Technology-agnostic metadata layer** JHeadstart defines the application in a metadata format that is not specific to any technology. Therefore, your application becomes more future-proof. For example, if you used JDeveloper 10.1.2 and JHeadstart 10.1.2 to build an application using Struts and UIX, you can upgrade to JSF and ADF Faces by regenerating your application using JHeadstart 10.1.3. The JHeadstart metadata model changed in version 10.1.3, but a 10.1.2 model will automatically be upgraded to the 10.1.3 format before you regenerate your application. The total effort of such an upgrade will depend on the number of post-generation changes you made. These changes will have to be re-applied manually if they are not natively supported in the new release.

Using JHeadstart

JHeadstart can be used in these two ways:

- **Build from scratch** In this method, you do not use or need to know Oracle Designer and Oracle Forms.
- **Build from Oracle Designer metadata** You can build ADF web applications by migrating them from existing Oracle Designer metadata. These metadata might already exist because you have previously generated your Oracle Forms applications from Designer. If your Oracle Forms applications are not defined in Designer, you can reverse engineer (also called "design capture") them into Oracle Designer, and then migrate them to ADF using JHeadstart.

This white paper discusses the second method in more detail in the section "JHeadstart Designer Generator." For now, we will concentrate on the first method, illustrated in Figure 1.

Developing J2EE applications with Oracle JHeadstart is a simple and iterative process. You start with creating the business layer. Next, you create the application metadata and generate a first-cut application. You continue refining the metadata and generate again until you are satisfied with the result. Finally, you can refine the application manually using the JDeveloper ADF design-time tools.

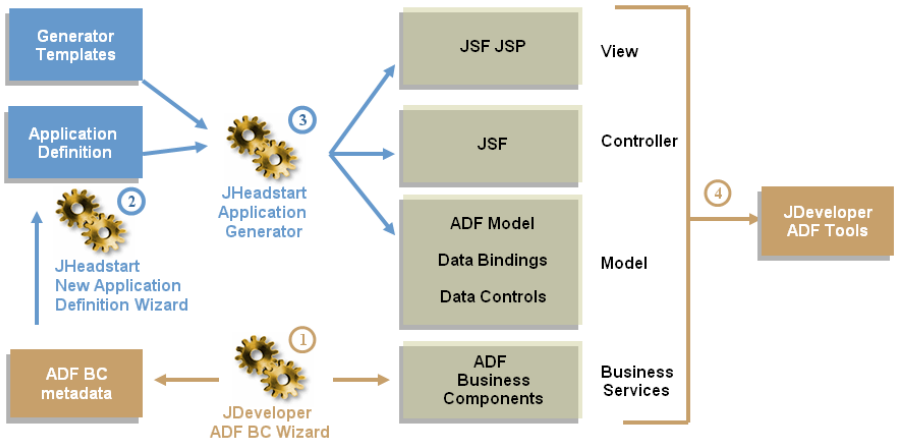

Figure 1: Building an application from scratch using JHeadstart

The high-level development process shown in this diagram follows:

1. Create the business service using ADF Business Components wizards in JDeveloper. This step is independent of JHeadstart.
2. Use the JHeadstart New Application Definition Wizard to create a first-cut of the *application definition*, the metadata file in XML format required to generate the application. Then, although it is not shown on the diagram, you would refine the metadata using the Application Definition Editor, and customize the generator templates using the JDeveloper code editor.
3. Generate the Model (data bindings), View, and Controller layer code using the JHeadstart Application Generator. This is a highly iterative process, where you refine the metadata and templates based on previous generation results. For an example of a generated page see Figure 2.
4. If the results from the JHeadstart generator do not fully match your functional requirements, you can enhance the generated pages using the JDeveloper ADF tools (visual editors, property inspectors, and drag-and-drop facilities). There are several ways to preserve post-generation changes, as we will discuss later.

Note: In addition to modifying code you generate using the visual and declarative editors, you will also likely write some Java code to provide specific functionality using the managed bean classes, and, perhaps, extending the JHeadstart framework classes. The documentation set included with JHeadstart will assist with these types of modifications.

While the main benefit of JHeadstart is in the Design and Build phases of your project, it can be a great help during analysis as well, particularly when you use an iterative, or Agile development approach. The New Application Definition Wizard, together with the ADF Business Components wizard allows you to create feature-rich, working prototypes in minutes. These prototypes help you validate user requirements gathered in an analysis workshop.

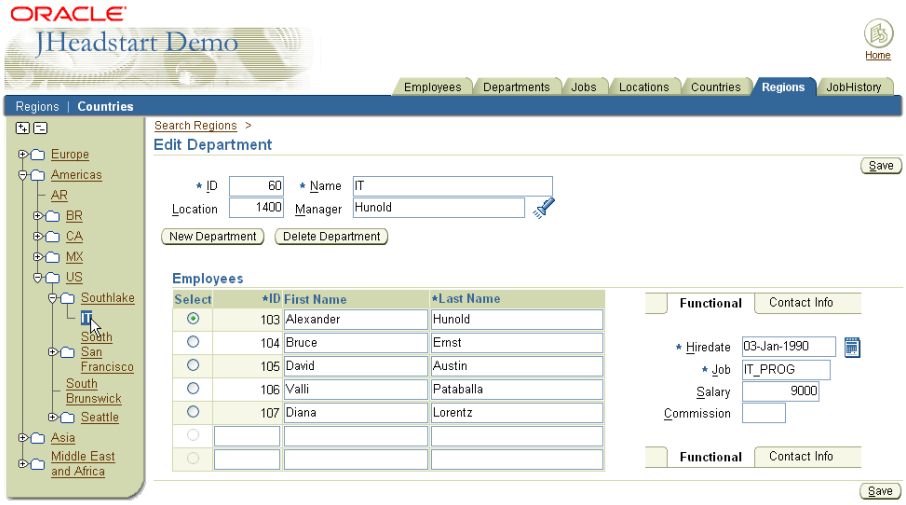

Figure 2: Example of a page fully generated by JHeadstart

Embedding JHeadstart in the development process increases your productivity, without jeopardizing the flexibility, maintainability, and openness of J2EE development using ADF. This is an important point to make, since many people have negative associations with code generators. Generated code can be hard to understand and tedious to modify when you need functionality not provided by the generator. These disadvantages do not apply to JHeadstart because it does not generate Java code. Rather than generating Java code, JHeadstart includes generic reusable runtime components that are configured and wired together through the generated faces-config.xml file and JSF JSP files.

To give you a more thorough understanding on how this works, we will discuss the following topics in more detail:

- **The JHeadstart application definition file** structure and how to create and refine it
- **The JHeadstart Application Generator** output and how to customize the result using generator templates.
- **The JHeadstart runtime** libraries and examples of generated screens

UNDERSTANDING THE APPLICATION DEFINITION

The main input to the JHeadstart Application Generator is the *application definition*, an XML file that defines the data collections to use, the page layout styles, query behaviors, transactional behaviors, and the user interface components to be generated.

This section discusses how to create and edit the application definition as well as describing the elements defined in this file.

Based on the ADF business components JHeadstart creates an initial set of metadata that can be used to generate your application. You can have a first-cut application in a matter of minutes, which allows for rapid prototyping.

Creating an Application Definition

You use the JHeadstart New Application Definition Wizard to create an initial application definition file. In this wizard you select an ADF Business Components application module, and specify default layout settings to be used for all groups as shown in the following wizard page:

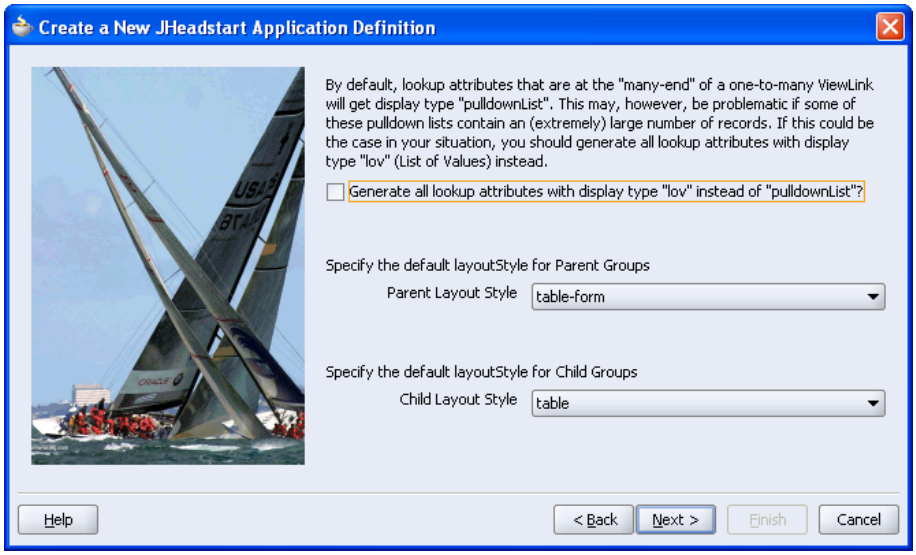

Illustration 1 Page in the JHeadstart New Application Definition Wizard

Results of the New Application Definition Wizard

The wizard will create an application definition file (visible in the Resources node of the navigator) that contains a group for each view object instance in the application module you select. Nested (parent-child) view object instances will be represented as nested groups. Dynamic domains for allowable values (with item *Display Type* as "dropDownList") or LOV groups (with the item *Display Type* as "lov" and a list of values element added to the item) will be created for "lookup" view links between view objects.

For example, the RegionCountriesFKLink view link represents the one-to-many link between Region.RegionId and Country.RegionId. If the application module contains a top-level view object instance named "Countries," the wizard will create a group named "Countries." In this group, the RegionId item will have a *Display*

Type of "dropDownList" or "lov" (with an associated LOV group or dynamic domain, respectively).

Note: You specify the generation of pulldown lists or LOVs with the checkbox "Generate all lookup attributes with display type "lov" instead of "dropDownList"?" as shown in the preceding illustration

If the application module contains a top level view object instance "Regions" and a nested view object instance "CountriesInRegion" using the same view link RegionCountriesFKLink , the wizard will create a master group "Regions" and a detail group "CountriesInRegion". In the CountriesInRegions detail group, the wizard will not assign the *Display Type* of the RegionId item as "lov" or "dropDownList" because the RegionCountriesFKLink is already used as the link to the parent group "Regions." Therefore, it would not make any sense to have a "lookup" to regions because the country is always accessed in the context of a region, and the link mechanism populates the RegionId value automatically for a new Countries detail row.

For example, a DeptEmpFKLink view link represents the link between Departments.ManagerId and Employees.EmployeeId. In this situation, Departments is a detail of Employees because Departments contains the foreign key on ManagerId. For the detail group, Departments, the wizard assigns a *Display Type* for the ManagerId as "lov" or "dropDownList" (with associated LOV group or dynamic domain, respectively). This allows the user to select a master Employees record to supply a value for the Departments.ManagerId attribute.

The Application Definition Editor

Refining and extending the initial metadata is straightforward. The JHeadstart Application Definition Editor includes a tree-structured navigator for easy access to the metadata objects, a context-sensitive property palette allows you to change the metadata properties with simple mouse clicks.

Although you can directly edit the raw XML of the application definition file, JHeadstart provides a properties editor, which you can invoke through the right-click menu on the project node in JDeveloper's navigator. If the project contains multiple application definition files, a submenu will appear that lists them all.

The left side of the Application Definition Editor, shown in Figure 3, contains the tree structure of the application definition elements, and the right side displays the properties of the currently selected node. All information required to drive the JHeadstart Application Generator is shown in this editor, which provides a complete overview of the application.

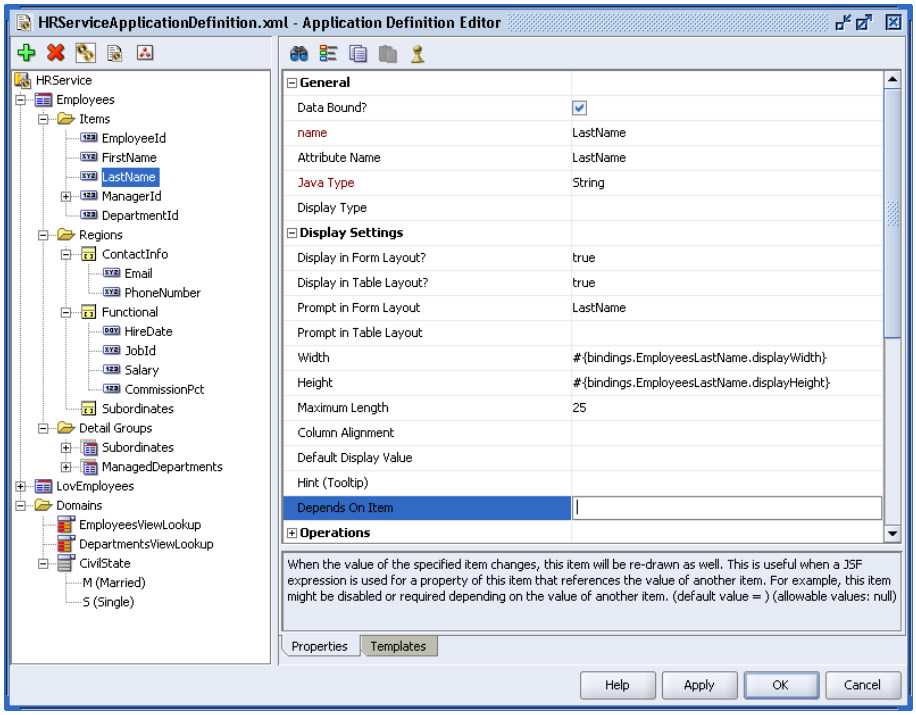

Figure 3: JHeadstart Application Definition Editor

The properties are logically grouped, and a description of the selected property is displayed at the bottom of the property inspector.

Main features of the Application Definition Editor include the following:

- **Copying and moving** elements using drag and drop. For example, you can drag and drop one or more items into an item region.
- **Switching between novice mode and expert mode.** In novice mode, advanced properties are hidden, as well as properties that currently do not apply because of values chosen for dependent properties. Expert mode shows all properties.
- **Multi-selecting properties to copy and paste property values** to other objects of the same type.
- **Displaying the appropriate editor** for a particular property as in the rest of JDeveloper. Depending on the property type, you might see a text input field, a checkbox, a dropdown list or pulldown list, or a file browse button.

Contents of the Application Definition

The application definition contains several elements that you can manipulate (as shown in the Application Definition Editor navigator in Figure 4): a service, groups, items, list of values (LOV), region containers, item regions and group regions, and domains. Each element has a number of properties.

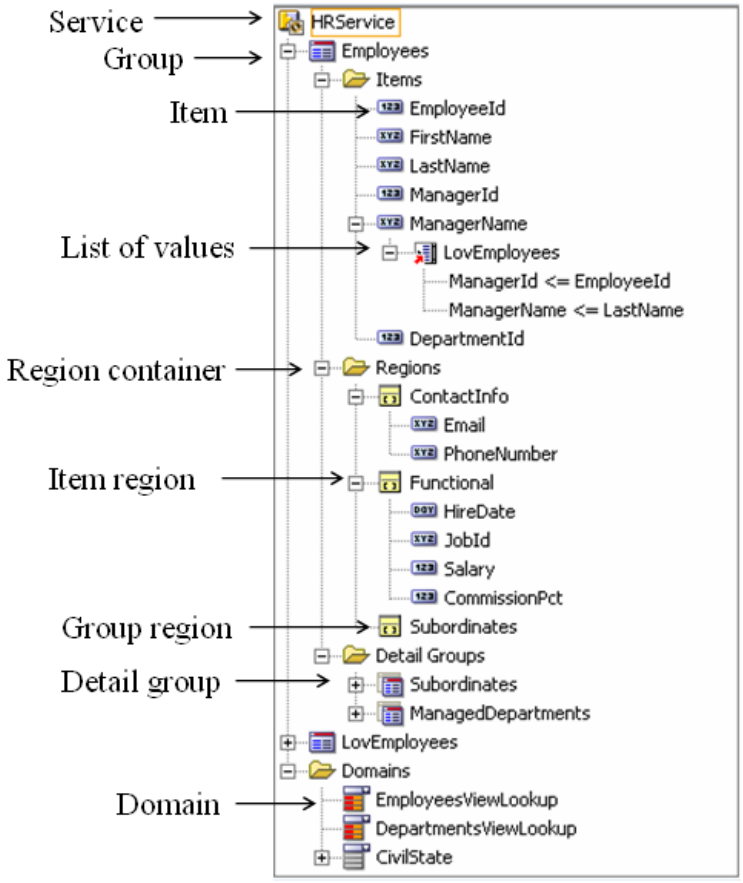

Figure 4: Application Definition Editor navigator showing metadata elements

In the Service level properties you define settings that apply to the application as a whole, like generation directories, date format, internationalization and security settings.

Service

The top-level node in the application definition is the *service* —a functional subsystem of the application. It includes logically related functionality on which a user performs tasks that are logically linked together. The service level contains definitions for, among other things, the ADF data control that will be used (the application module in the case of ADF Business Components). You use the same considerations to create one or more application definition files as you do to create one or more application modules in the Business Tier.

The service includes properties to specify file locations, general UI settings (for example, date and date-time formats), security, and internationalization settings as shown in an excerpt of the service-level Property Inspector window here:

Identification	
Name	HRService
Description	Application Definition for HRService
Generator Flavours	
View Type	ADF Faces
File Locations	
Faces Config	/WEB-INF/faces-config.xml
UI Pages Virtual Directory	/WEB-INF/page/
Pages Common Virtual Directory	/common/
Stylesheet Relative Url	jheadstart/css/jspStyles.css
Java	
View Package	oracle.hr.view
Data Control	HRServiceDataControl
Data Control Implementation	oracle.hr.model.service.HRService
Set Application Module Superclass to JhsApplicationModuleImpl	☑
Supporting Files	
Templates Base Directory	
UI Settings	
Overall Layout Style	Menu Style
Show Hint Text?	☐
Date Format	dd-MMM-yyyy
DateTime Format	dd-MMM-yyyy HH:mm
Set Date(time) Formatter on EntityObject Attributes?	☑
Default Display Width	60
Unselected Label in Dropdown List	
Allow Partial Last Page in ViewObject Page Iteration	☑
Authorization	

Illustration 2: Service level properties

The UI Settings property *Overall Layout Style* with allowable values "Menu Style" and "Wizard Style" defines whether all generated pages should be accessed via a common menu bar, or linked together using a wizard style layout with Back and Next buttons to navigate through the pages.

A group is linked to a data collection that can be displayed on one or more pages depending on the layout style and search settings. The allowable operations (insert, update, delete) can be set on the group, and can be made conditional by specifying the user roles required for an operation. Groups can be nested to create master-detail pages

Group

A service consists of one or more groups. A *group* is tied to one data collection within the data control, which maps to a view object instance when using ADF Business Components. The group contains properties to set the layout, the allowable operations (insert, update, delete) and to specify the query and search behaviors.

A group is similar to the concept of an Oracle Forms block, however, a group definition can result in multiple pages, whereas an Oracle Forms block usually appears on a single canvas. The number of pages generated for a group depends on

the *Layout Style* property (form, table, table-form, select-form, tree, tree-form, parent-shuttle, intersection-shuttle) and the *Advanced Search* setting (samePage, separatePage or none). Groups can be nested to represent parent-child relations, and child groups are called *detail groups*.

The *Same Page* property for detail groups can be used to indicate whether the detail group should be generated on the same page as its parent. The following illustration shows some of the properties for a group:

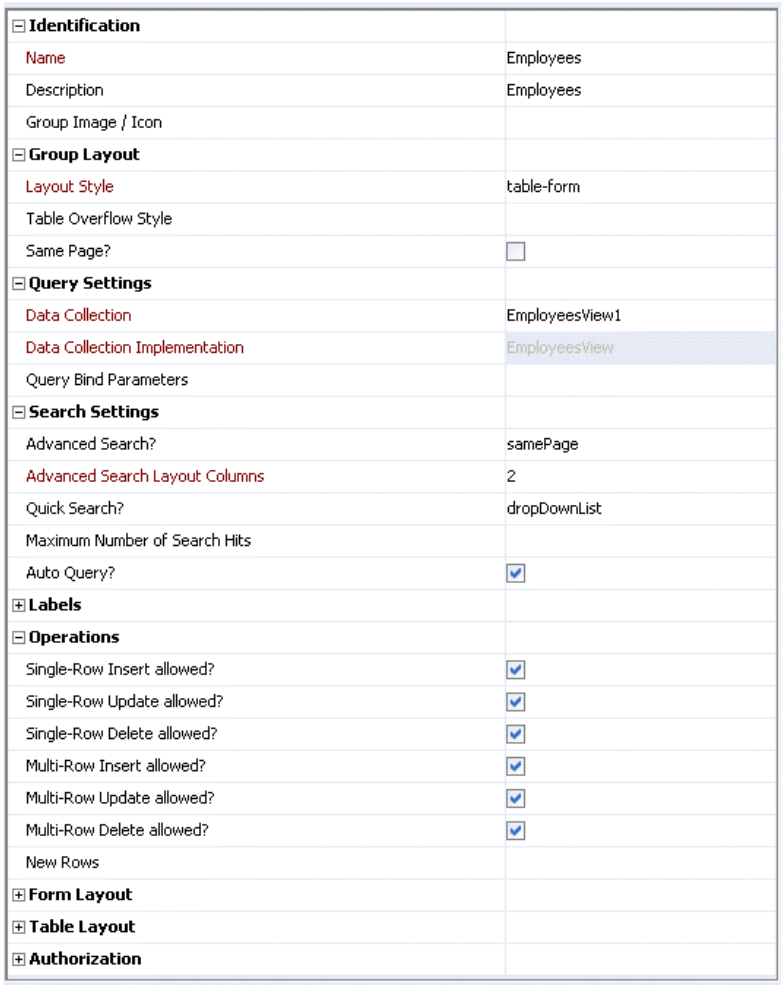

Illustration 3: Group level properties

Comparing this to an Oracle Forms module defined through Oracle Designer, you would typically create one JHeadstart group for each top-level module component you define in Designer. For detail module components in a master-detail relationship, you would use detail groups in JHeadstart.

Item

A group contains one or more items that can be displayed on the page. You can specify a wide variety of properties on the item including display settings, operations, validation and query settings.

A group contains one or more *items*, components that represent single data elements. Items can be databound or unbound. *Databound items* are based on an attribute of the data collection associated with the group. Unbound items can be used to generate buttons, hyperlinks or "control" fields. The *Display Type* property of an item defines the user interface widget that is generated, for example text input, dropdown list, checkbox, radio group or file download link.

Other properties that can be set include Prompt in Form Layout**,** Prompt in Table Layout, Display in Table Layout, Display in Form Layout, Width, Maximum Length, Height, Required, Insert Allowed, Update Allowed, Default Display Value, Read Only JSF Expression, and Disabled as shown in the next illustration. Where appropriate, you can use a JSF expression as the value of a property. For example, you can make an item read-only based on the users' security roles.

General	
Data Bound?	☑
name	LastName
Attribute Name	LastName
Java Type	String
Display Type	textInput
Display Settings	
Display in Form Layout?	true
Display in Table Layout?	true
Prompt in Form Layout	LastName
Prompt in Table Layout	
Width	#{bindings.EmployeesLastName.displayWidth}
Height	#{bindings.EmployeesLastName.displayHeight}
Maximum Length	25
Column Alignment	
Default Display Value	
Hint (Tooltip)	
Depends On Item	
Operations	
Insert Allowed?	true
Update Allowed?	true
Read Only JSF Expression	
Disabled	
Validation	
Required?	#{bindings.EmployeesLastName.mandatory}
Validator Binding	
Regular Expression	
Regular Expression Error Message	
Query Settings	
File Upload/Download Settings	

Illustration 4: Item level properties

The JHeadstart Application Definition Editor contains a button to synchronize the group items with the underlying data collection. New attributes added to the data collection will be added as databound items.

Items can be grouped into regions on the page. Regions can be nested and the layout style of the parent region determines how the regions are displayed relative to each other: horizontally, vertically or stacked.

Region Container, Item Region and Group Region

Items can be grouped in an item region — a logical grouping of attributes with an optional title. Multiple regions can be grouped in a *region container*. The region container *Layout Style* property specifies the layout of the regions in the container: horizontally, vertically, or stacked. Region containers can be nested to generate nested stacked regions. A region container can also contain a *group region*—a region container for a single detail group. A group region has one property *Detail Group Name*, which specifies the name of a detail group that should be placed in the region. Items within the detail group will be placed in the location of the region. Figure 3 shows item regions for ContactInfo and Functional as well as a group region for Subordinates.

Without JHeadstart, you would use container components such as af:panelHeader and af:panelForm to groups input components or other container components. The container component supplies layout behavior such as vertical or horizontal stacking.

Note: Since JHeadstart generates prebuilt page layouts, it is worthwhile examining the pages to see how the container components are nested. This can help make you more aware of how container components may be used if you need to make changes to JHeadstart pages or to your own custom pages.

JHeadstart generates sophisticated list of values (LOV) pages. You can return multiple items from the LOV to the base page, and the LOV can be used to validate the value in the LOV item. The value will be auto-completed if only one matching LOV row is found.

List of Values

You can associate a list of values (LOV) with an item. In JHeadstart, a *list of values* displays values from a view object in a popup window. The *LOV Group Name* property of the LOV specifies the name of the group used to generate the list of values page. Only groups that have the property *Use as List of Values* set to "true" can be chosen. A list of values element contains one or more *return value* elements. A *return value* defines a lookup-base attribute pair—that is, the attribute of the selected row in the LOV group's data collection and the attribute of the current row of the base group data collection to which the value will be copied.

JHeadstart can also generate multiple selection, using LOVs for validation, and LOVs on read-only attributes.

A powerful property of the list of values element is *Use Lov for Validation,* shown in the LOV property inspector in the following illustration. When checked (true), the LOV will automatically appear when the user tabs out of the field with the LOV if the value entered matches none or multiple rows in the lookup data collection. When the value matches exactly one row, the LOV is not shown and the value will be auto-completed if required. This type of validation and auto-completion is a well-known feature to Oracle Forms developers.

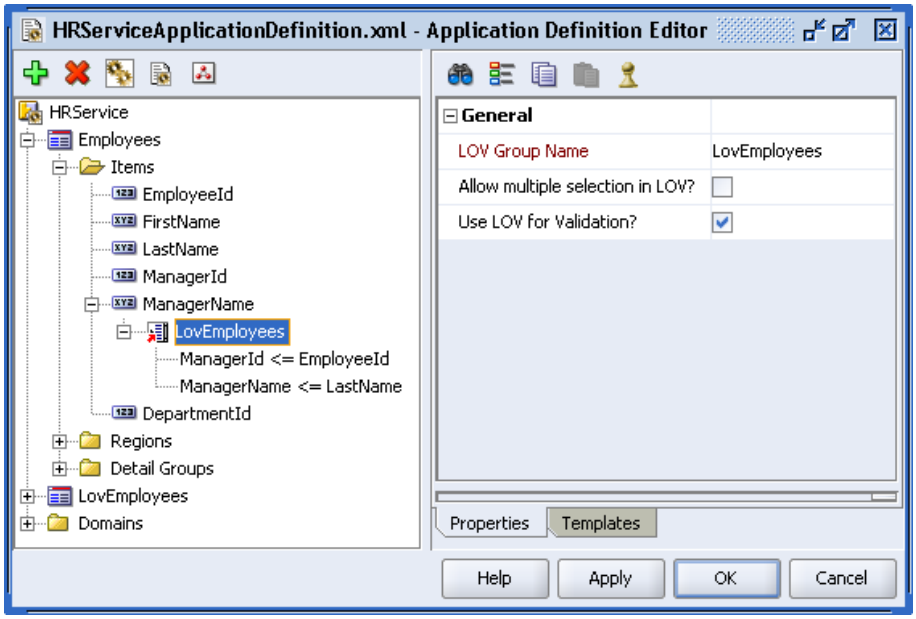

Illustration 5: Defining a list of values with auto-completion

Multi-select LOV's (where the user can select more than one value) can be generated by checking the *Allow multiple selection in LOV?* property. When this property is checked, and the user selects multiple rows in the LOV, new rows will be created in the base page. A typical example is a page containing a master order and order lines details; on this page, a multi-select LOV for products will create new order line records, one for each selected product.

Domain

The *domain* element (shown in the next illustration) specifies a static or dynamic list of allowable values. When the *Domain Type* property is set to "dynamic" you can use the *Data Collection* property to specify the data source of the allowable values. When the *Type* property is set to "static", you can specify allowable value child elements (in the same way you add allowable values to a Designer domain). The allowable value element has two properties *Value* and *Meaning*. Domains can be used to generate items with a type of radio group, dropdown list, or checkbox. Domains are not used for list of values, as they have no properties to specify how the LOV page should look and behave. List of values are defined using the Group element as explained before.

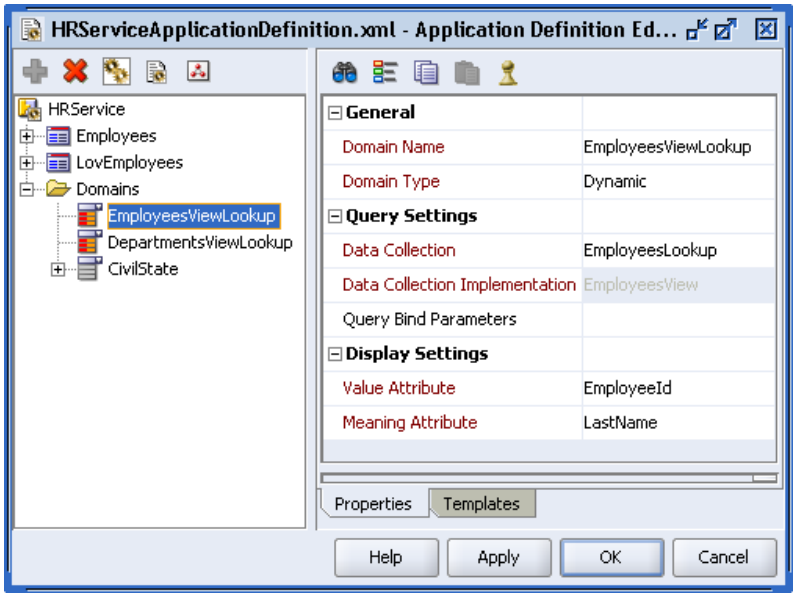

Illustration 6: Defining Domains

You associate a domain with one or more items through the *Domain* property of the items. The *Display Type* property of the item determines if the list of allowable values is presented to the user as a radio group, a checkbox, a dropdown list, or a list box (text list).

RUNNING THE JHEADSTART APPLICATION GENERATOR

You can run the JHeadstart Application Generator (JAG) in two ways: using the iconic button in the Application Definition Editor, or using a right-click menu option on the application definition file in the JDeveloper navigator.

This section discusses the outputs of the JAG in more detail and describes how you can preserve changes to these outputs made after generation. First we will briefly introduce you to the powerful concept of generator templates.

Generator Templates

The content of the generated pages is completely driven by generator templates. By defining custom templates you have full control over the generated output, you can customize it any way you want.

The content of the generated JSP pages and faces-config.xml file is driven by templates. These templates can contain static content that will be included as is in the generated page, as well as dynamic content. Dynamic content is defined using the *Velocity Template Language* (VTL). *Velocity* is an open source Java-based template engine, which is an effort of the Apache Jakarta Project (jakarta.apache.org/velocity). It permits anyone to use a simple yet powerful template language to reference objects defined in Java code. When running the JAG, JHeadstart creates Java objects for the various elements of the application definition, and calls the Velocity Template Engine to resolve the VTL constructs in the various generator templates.

JHeadstart ships with a large set of default templates for generating the faces-config.xml file, menus, overall page layout, search regions, item regions, various group layout styles (tree, form, table, parent-shuttle, intersection-shuttle, select list), and all item types. The JHeadstart page-level templates are comparable to the Oracle Designer Forms Generator templates; all JHeadstart lower-level templates roughly correspond to objects specified in the Oracle Forms object library. Like the Oracle Designer Forms Generator, you can create custom generator templates, and configure the JAG to use your custom template instead of the default one. Once you become familiar with the template structure and VTL, you will be able to customize the generator outputs any way you want.

Note: You can find documentation including a reference of all VTL constructs at http://jakarta.apache.org/velocity/docs.

Generator Outputs

The JHeadstart Application Generator is capable of generating the following types of output:

- **JSF JSP Pages** in XML format (.jspx files)
- **faces-config.xml file** for the JSF Controller.
- **PageDef files** containing the ADF Model bindings for the generated pages.
- **Resource bundles** for internationalization.

JSF JSP Pages

The structure of a generated JSF JSP page is identical to the pages that you create manually using drag and drop. This means that you can open a generated page in the visual editor, and add or modify functionality manually. See the section "Making and Preserving Post-Generation Changes" for more information.

faces-config.xml

JHeadstart generates managed bean definitions into the faces-config file to configure the generic JHeadstart bean classes for usage in a specific page. The bean classes provide behaviors like advanced search, multi-row insert and delete and shuttle functionality.

The main contents generated into the faces-config.xml file are the navigation rules and the managed bean definitions. The managed beans, used to hold code for the application's pages, are defined in the faces-config.xml file.

Classes from the JHeadstart runtime library are used as the managed bean classes. One library class can serve as the managed bean for a page because the classes are highly configurable. Each instance of the managed bean class is configured for usage in a specific page. For example, many pages in the generated application might offer a quick and/or advanced search region (as explained and shown later in the "Examples of Generated Screens" section).

For each search region in each page, a managed bean definition such as the following is added to the faces-config.xml file, reusing the same generic JhsSearchBean class:

```
<managed-bean>
  <managed-bean-name>searchEmployee</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.bean.JhsSearchBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>bindings</property-name>
    <value>#{data.EmployeeTablePageDef}</value>
  </managed-property>
  <managed-property>
    <property-name>searchBinding</property-name>
    <value>#{data.EmployeeTablePageDef.advancedSearch}</value>
  </managed-property>
  <managed-property>
    <property-name>dataCollection</property-name>
    <value>EmployeesView1</value>
  </managed-property>
  <managed-property>
    <property-name>maxQueryHits</property-name>
    <value>50</value>
  </managed-property>
</managed-bean>
```

As you will notice, the JhsSearchBean JHeadstart runtime class acts as the managed bean for the search region in the EmployeeTable page. Various properties (parameters) are passed to this class using the managed property elements. In this way, you can reuse the JhsSearchBean for search regions in different pages and

define page-specific values in the managed-property elements to customize its behavior.

This same concept is used to implement other JHeadstart runtime features like the LOV functionality, the shuttle functionality, query parameter binding, tree forms, and multi-row insert and delete. All these features rely on generic classes included in the JHeadstart runtime library; the classes are configured as managed beans in the faces-config.xml. This architecture makes it easy to customize or extend the default behavior of the JHeadstart managed beans: you can extend a managed bean class and to ensure your custom class is used, you customize the generator template used for the specific managed bean definition.

PageDef files

ADF creates a PageDef file when you drag and drop objects from the DataControl palette to your page. The PageDef file holds the executables and bindings of your page, and is required to run your pages. JHeadstart-generated pages work in the same way. For each page, the JAG will create a PageDef file containing executables and bindings based on the information in the application definition.

Resource Bundles (Message Bundle)

The JHeadstart Application Generator creates a resource bundle (message bundle) for text used on the screens. You can use different resource bundles to hold text translated into different languages. You specify the name of the resource bundle in the service-level property *NLS Resource Bundle*. The *Resource Bundle Type* property defines whether the generator creates a property file or a Java class.

The JAG writes text into this file for boilerplate text and for button labels and page header titles. You can use the service-level checkbox *Generate NLS-enabled Prompts and Tabs* to also load prompts and tab names, and display titles. This is useful if you need to translate this text, as well. The Internationalization section of the service node in the application definition appears here:

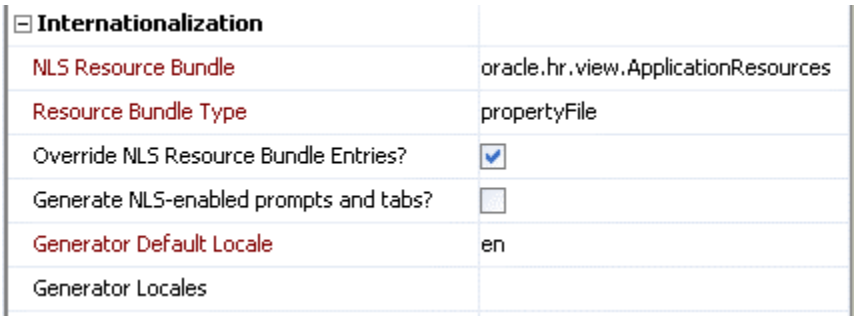

Illustration 7 Defining Internationalization properties

Using the *Generator Locales* property, you can specify which locales are supported by your application. A resource bundle will be generated for each locale.

Making and Preserving Post-Generation Changes

If you have used the capabilities of the JHeadstart Application Generator to its full extent, you can use the JDeveloper ADF tools to add additional functionality to your pages. There are different ways you can preserve these post-generation changes. One technique is to make these changes in a custom template, which allows you to keep on generating!

The JHeadstart Application Generator is a powerful generator, and by customizing the generator templates, you have a lot of control over the functionality and look and feel of the generated application. It is probably not realistic for most web applications to expect 100% generation (as some shops were able to attain with Oracle Designer Forms Generator), and too much focus on trying to generate a feature that is not supported out-of-the-box might be counterproductive.

If you have experience using the Oracle Designer Forms Generator, you might remember how hard it can be to make the fields layout properly. Various generator preferences and the *Relative Tab Stop* property can be used to govern the layout. Real Oracle Designer Forms Generator gurus can generate any layout they want, using Oracle Forms Builder to change the layout of the generated is much faster.

The same principle applies to JHeadstart. The more skilled you are in JHeadstart, the more likely you are to stretch the capabilities of the JHeadstart Application Generator. But remember, JHeadstart is there to help you work faster, the moment you get the feeling that JHeadstart hinders you in implementing functionality you should abandon generation and build the remaining functionality manually using the visual editors, code editors, and drag and drop from the Data Control Palette.

Some people just stick to generation because lack an understanding of how to build things manually using ADF. You may however have acquired enough knowledge on how to work with ADF. All this knowledge is also applicable when performing post-generation changes. After all, the runtime architecture, the JSP pages, the PageDef files, and the faces-config.xml file produced by JHeadstart, are all very similar to applications that are hand built.

Ideally, you start applying post-generation changes when the data model and the system requirements are fairly stable. But we do not live in an ideal world, and certainly now with the increasing popularity of agile development approaches, it is hard to tell when the system functionality has become stable (if it ever does).

Options for Preserving Post-Generation Changes

More often than not, you will be confronted with changing requirements that are easily implemented using the JHeadstart Application Generator. However, what happens if the requirements change and you have already made post-generation changes? You basically have three options:

- **Do not regenerate the parts** of the system that have been changed
- **Regenerate the entire system** and re-apply the post-generation changes afterwards.
- **Define all your post-generation changes in custom templates**, so nothing gets lost after regenerating.

Turn Off Generation for Modified Files

The first option is supported by a number of properties called *generator switches* that govern the files or parts of files that will be regenerated. These switches appear at these levels:

- **The service level** offers coarse-grained switches you can set to prevent regeneration of the faces-config.xml file, the NLS resource bundle(s), and the menu bar.
- **The group level** supplies switches to prevent regeneration of group-specific JSF JSP pages, the PageDef files, and the managed beans and navigation rules in the faces-config.xml file.

Custom managed beans and navigation rules that you add manually to the faces-config.xml file will never be removed or overridden, regardless of the generator switch settings. This also holds true for custom executables and bindings that you add manually to the PageDef file or that are added when you drag and drop onto the visual editor.

Regenerate the Entire System

The second option of re-applying post-generation changes sounds less attractive, but might be useful when the generator switches are too coarse-grained and the post-generation changes are quickly and easily reapplied. For example, you have a master-detail page where the detail UI block needs to be regenerated because of new requirements, and the master "block" has post-generation changes. In this case, it might be much faster to regenerate the whole page and re-apply the post-generation changes to the master block. Of course, this approach requires discipline: you need to carefully document your post-generation changes step by step.

Define Post-Generation Changes in Custom Templates

If you have chosen the second option, and you find yourself re-applying the same post-generation changes over and over again, you might actually go for a third option that merges the first two options. JHeadstart provides an extremely fine-grained template mechanism, that is, each UI component on the page has its own template that you can customize. Using the example explained before, you could create a custom template that only contains static content: the code of the master "block" including the post-generation changes. Using the Templates tab of the Application Definition Editor (shown in Figure 3), you can configure JHeadstart to use this custom template for the master block and when you then regenerate the page, the code in the custom template is simply copied into the generated page. This technique is the holy grail of 100% generation: you can move any post-generation change into a custom template if you like. Experiences in real-world projects indicate that this approach works very well, particularly in a RAD environment with customized modules that are expected to change often.

Note: The JHeadstart Web Log contains an interesting post about documenting post-generation changes. Although a bit out-of-date (the post was written for the JHeadstart 10.1.2 release compliant with JDeveloper 10.1.2), the technique described is still useful. The post can currently be found at http://www.orablogs.com/jheadstart/archives/001391.html.

RUNNING THE GENERATED APPLICATION

The generator outputs together with the JHeadstart runtime provide a fully functional application.

After you have run the JHeadstart Application Generator you get a fully functional and feature rich application. Check out the examples shown in this section.

Examples of Generated Screens

Rather than providing a long list of feature descriptions, we will show you a couple of generated screens and describe the generator settings that were used to create these screens. This will give you a feeling for what is feasible with JHeadstart, as well as how to use the JHeadstart Application Definition to specify these features.

Advanced Search, Editable Table, and Inline Overflow

Figure 5 shows a screen containing a search region, multi-row editable table, details region, and pulldown lists. This section describes how these features were defined.

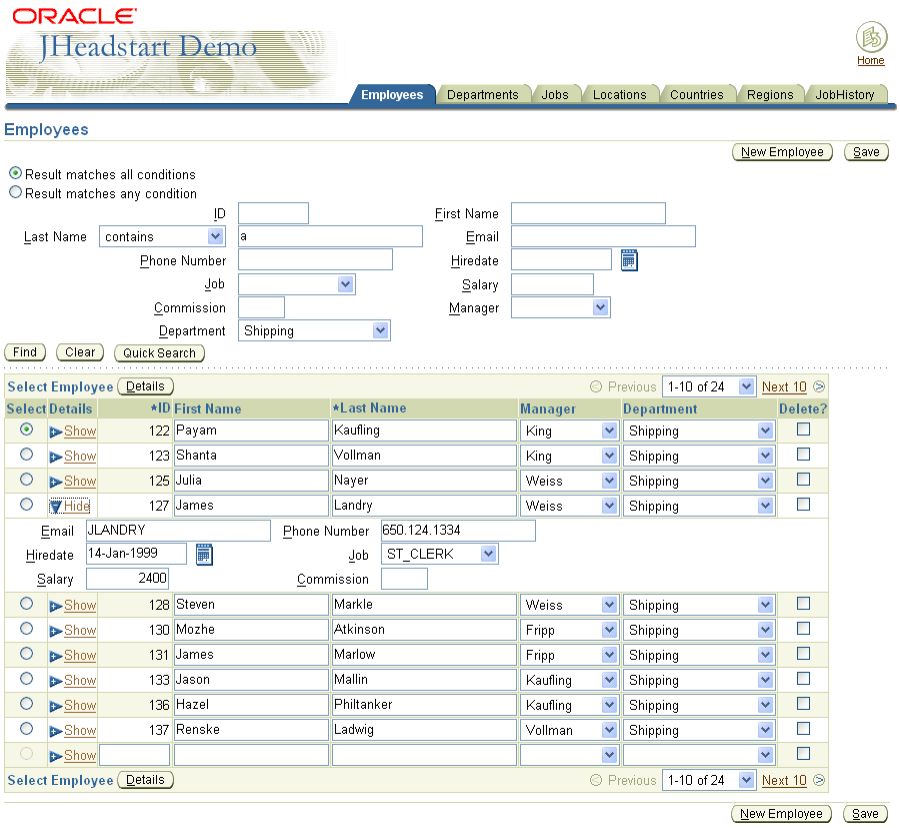

Figure 5: Generated page with advanced search, editable table, and inline overflow

Search Region

The search region is placed on the page because the *Advanced Search* property of the group is set to "samePage". Items displayed in the advanced search have the *Show in Advanced Search* checkbox checked. The dropdown list in the search field *Last Name* allows the user to set the type of SQL operator and wildcard usage in an intuitive way; this behavior is generated because the item property *Query Operator* is set to

"setByUser". The Quick Search button is generated because the group *Quick Search* property is set to "dropDownList". When clicking this button, the advanced search region will be hidden, and the quick search region will be displayed as shown here:

Illustration 8: Quick Search

The quick search field displays differently depending on the quick search item. For example, when the quick search attribute is set to Department, a dropdown list with all departments will be displayed instead of a text input field.

Multi-row Editable Table

The table is generated because the group *Layout Style* is set to "table-form". The Details button will display the Employee form page, which will show the selected employee record in single-row format. The table displays 10 rows at a time because the group *Table Range Size* property has been set to "10". The rows are updateable because the group checkbox *Multi-Row Update Allowed* is checked. The delete checkbox is generated because the group checkbox *Multi-Row Delete Allowed* is checked. The new row at the bottom is generated because the group checkbox *Multi-Row Insert Allowed* is checked and the group property *New Rows* is set to "1". The ID field in the table is only editable in a new row because the item property *Insert Allowed* is set to "true" and *Update Allowed* is set to "false".

Show/Hide Icons and Employee Details

The Show/Hide icons, and the employee details that are subsequently shown when clicking the Show icon are generated because the group *Table Overflow Style* property is set to "inline". The items displayed in the table row have the item property *Display in Table Layout* set to "true". The items displayed in the inline overflow area have that property set to "false" and property *Display in Table Overflow Area* set to "true".

Pulldown Lists

The items rendered as pulldown lists have the *Display Type* property set to "dropDownList", and an associated dynamic domain element with a *Data Collection* property that specifies the data shown in the dropdown list.

Parent UI Block with Stacked Children, List of Values, and Shuttle

The screen shown in Figure 6 contains a form-style block, shuttle control, stacked detail groups, and a list of values. It was generated with the settings described next.

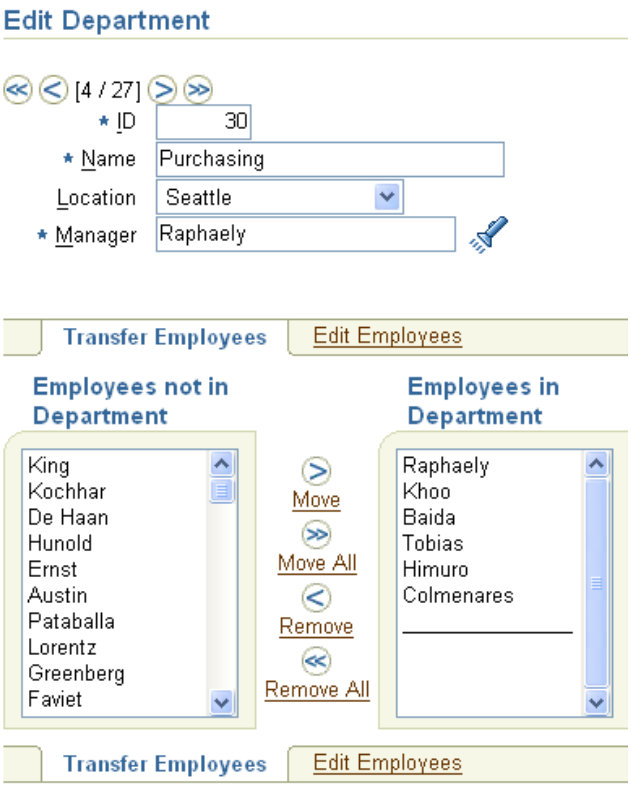

Figure 6: Generated page with stacked detail groups, list of values, and shuttle

Form-Style UI Block

The parent group Department *Layout Style* property is set to "table-form". The page shown represents the form layout that displays after selecting a record on a table page and clicking Details.

Shuttle Component

The detail group TransferEmployees has its *Layout Style* property set to "parent-shuttle" and the *Same Page* checkbox checked. When saving the changes, detail rows selected on the right side of the shuttle component in the parent group (Departments) will cause the foreign key value of a detail row (Employees) to be updated to refer to the selected parent row; the foreign key value for detail rows that have been deselected (moved to the left shuttle area) will be nullified. JHeadstart also supports an *intersection shuttle*, which will insert and delete entries in an intersection table.

Edit Employees UI Block

The detail group EditEmployees, not shown in Figure 6, has *Layout Style* set to "table" and the *Same Page* checkbox checked.

Stacked Detail Groups

The detail groups (that form the tabs) are stacked into tabs because the master group Departments has a *Region Container* named "Regions" with *Layout Style* set to "Stacked". This region container includes two detail group regions as shown in the following illustration. The *Detail Group Name* property of the group region is set to the detail group inside the region (shown next for TransferEmployees).

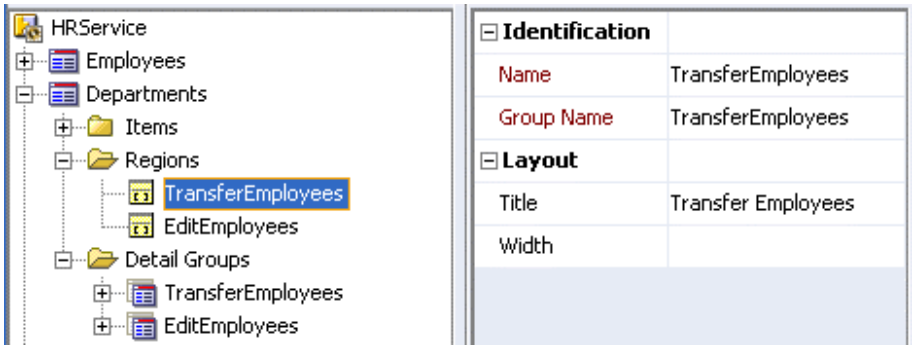

Illustration 9: Defining stacked groups

List of Values

The Manager item displays an LOV icon that invokes a separate list of values window because its *Display Type* property is set to "lov". The data shown in the LOV window and the query capabilities in the LOV are defined through the group element that supplies the values for the LOV. This group is associated with this item using the property *LOV Group Name* for the associated list of values.

Tree, Form, and Detail Table with Stacked Overflow

The screen shown in Figure 7 contains a tree component, a table area, and a stacked overflow area. These features were defined using the settings described in this section.

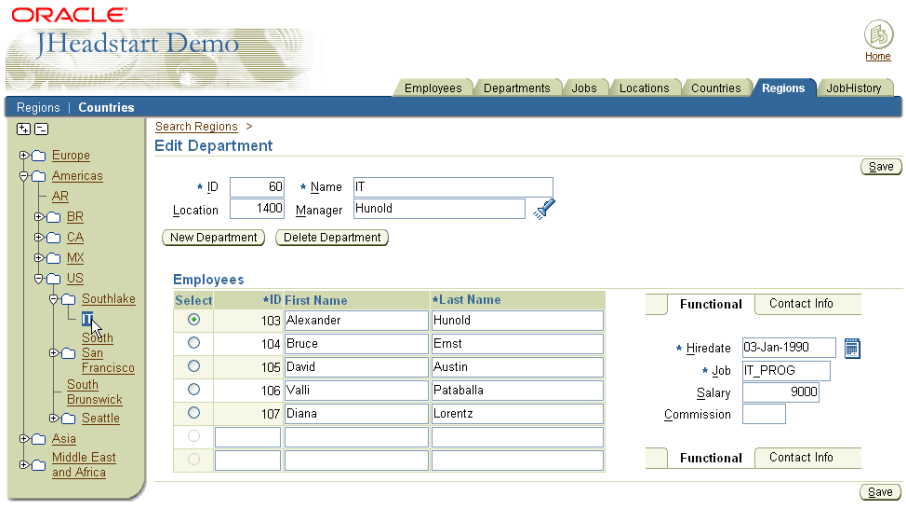

Figure 7: Generated page with a tree, form, and detail table with stacked overflow right

Tree Component

The group structure is nested five levels deep to represent Regions, Countries, Locations, Departments, and Employees. All groups except Employees have the *Layout Style* set to "tree-form". The *Layout Style* of the Employees group is set to "table" and the *Same Page* checkbox is checked.

Overflow

The *Table Overflow Style* property of the Employees group is set to "right". The overflow area shows items you do not want to display in the table because they would make the table too wide. Only EmployeeId, FirstName and LastName items have the property *Display in Table Layout* set to "true". All other items have this property set to "false", and the property *Display in Table Overflow Area* set to "true".

Stacked Overflow Tabs

The items in the overflow region appear stacked because the Employees group has a RegionContainer named Regions with the property *Layout Style* set to "stacked". This RegionContainer contains two ItemRegions with the *Title* property respectively set to "Functional" and "Contact Info".

The Role of the JHeadstart Runtime

The JHeadstart runtime consists of a set of Java classes (the sources are included), a JavaScript library, and a Cascading Style Sheet that together provide application functionality for all tiers of a J2EE web application. The JHeadstart runtime provides you with a host of ADF best practices as described in the book "Oracle JDeveloper 10g for Forms & PL/SQL Developers : A Guide to Web Development with Oracle ADF" and on Oracle's Technology Network website (otn.oracle.com).

Some examples of runtime components and the features they provide follow:

- **The JhsApplicationModuleImpl class** extends the standard ADF Business Components ApplicationModuleImpl class. It includes generic methods to perform advanced searches on a view object, to execute a query on a view object when bind parameter values have changed, and to handle shuttle selections.
- **The JhsPageLifecycle class** extends the default ADF PageLifecycle class to add additional behavior to standard ADF operations, for example:
  - In the onCreate() method, default display values, which you can define as JSF expressions in the application definition, are set on the new row.
  - In the onDelete() method the deletion is auto-committed, as the end user would expect when clicking a Delete button.
  - In the onCommit() method, user feedback similar to that in Oracle Forms will be displayed. For example, the user will see either a "No Changes to

save" or "Transaction Completed Successfully" message as a result of a save operation.

- In onRollback() method, the row currency and range start positions of all iterator bindings are restored to the values prior to the rollback. This means that query results will be displayed with the same current row and sort order as before the operation.

- **The JhsListOfValues class** is used as a JSF managed bean that provides LOV features like multi-select, copying multiple values back to the base page, and using the LOV to validate the value entered in the field that has the LOV attached.
- **The ReportingUtils class** enhances standard ADF error reporting. Container exceptions are filtered out, and only error messages that make sense to the user are displayed. Database constraint violations are handled gracefully, allowing the developer to specify a user-friendly error message in the resource bundle using the constraint name as the key.
- **The alertForChanges() function** in the JHeadstart JavaScript library displays a JavaScript alert (shown in the next illustration)when the user abandons a page that has unsaved changes.

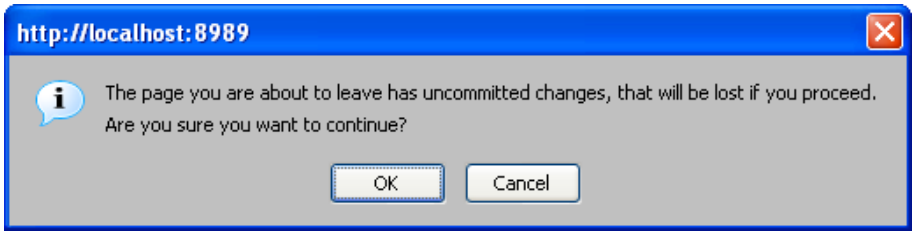

Illustration 10: Uncommited changes warning

This list is not exhaustive by any means; it is just intended to give you an idea of the types of functionality provided by the JHeadstart runtime.

JHEADSTART DESIGNER GENERATOR

If you are familiar with generating Oracle Forms applications using Oracle Designer, you may have noticed in the preceding discussion that the metadata structures you can specify in the JHeadstart Application Definition are very much inspired by the Oracle Designer metadata model. Some properties even have identical names, such as the group properties *Table Overflow Style* and *Descriptor Attribute*.

JHeadstart allows you to reuse the Oracle Designer metadata by converting it to the XML format that can be read by the JHeadstart Application Generator. This allows you to generate applications from the metadata in the Oracle Designer Repository, and to migrate Oracle Forms applications to J2EE.

A major difference between Oracle Designer and JHeadstart is the storage format: JHeadstart uses an XML file, and Oracle Designer uses relational tables. The *JHeadstart Designer Generator (JDG)* allows you to convert the Oracle Designer metadata to the XML format required by JHeadstart and it creates the ADF Business Components metadata. This effectively allows you to reuse Oracle Designer metadata to generate ADF applications. It is a two-step process as shown in Figure 8.

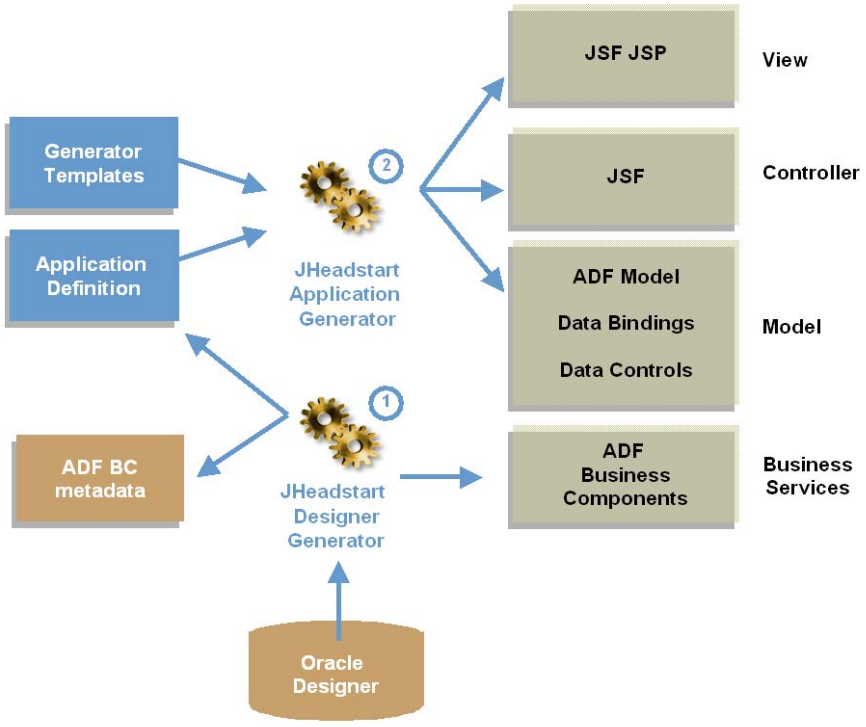

Figure 8: JHeadstart Designer Generator migration process

In the first step, you run the JHeadstart Designer Generator to create the ADF Business Components, the ADF BC metadata, and the Application Definition. The second step is to run the JHeadstart Application Generator to create the Model, View, and Controller files required to create a fully functional ADF web application.

The business components are created as follows:

- **Designer table definitions** are implemented as ADF BC entity objects.
- **Designer module components** are implemented as ADF BC view objects
- **Designer modules** are implemented as ADF BC application modules.

You will typically run the JDG only once for a set of Designer modules. If for some reason you change the definitions in Oracle Designer after running the JDG, you will need to rerun the JDG. After running JDG, if you make changes to a group in the application definition file that you do not want to lose when rerunning the JDG, you can set the check the *JDG Protected* checkbox at the group level.

Note: JHeadstart includes an "Oracle JHeadstart Designer Generator Reference", which lists how each metadata element in Designer is converted to its JHeadstart equivalent.

Design Capturing Oracle Forms in Designer

If you generated your Oracle Forms with Oracle Designer you can migrate your Oracle Forms to J2EE right away. If you manually built your Forms using Forms Builder, you should use the Design Capture facility in Oracle Designer first.

You might wonder how the JHeadstart Designer Generator can help you when you have not used Oracle Designer to generate your forms, or you did generate your forms but customized them heavily afterwards using Oracle Forms Builder. If this is the case, you can first design capture (reverse engineer) your database and Oracle Forms definitions into the Oracle Designer repository by using Designer's Design Capture Wizard. This wizard creates the metadata in the repository according to the following list from the Oracle Designer online help system:

- **A new module definition** for the form being captured. If the module definition already exists, you can merge the results of the design capture with the existing form definition.
- **Window definitions** for windows in the form.
- **Module components** for blocks in the form.
- **Base table usages** for base table blocks in the form.
- **Lookup table usages** when capturing generated forms containing LOV usages
- **Bound and unbound items** for all items in the form except control block items displayed on vertical or horizontal toolbar canvases.
- **Item groups** for items on different tab pages of a tab canvas in the form
- **Module arguments** for each parameter in the form.

Note: In the preview and first production release of Oracle JHeadstart 10.1.3 the JHeadstart Designer Generator will not be included. There is however a workaround. Use the Oracle JHeadstart 10.1.2 version to create the JHeadstart metadata of your application from Oracle Designer. Migrate this metadata to Oracle JDeveloper 10.1.3 and use the JHeadstart Application Generator 10.1.3. to generate the application.

MIGRATING ORACLE FORMS APPLICATIONS

Migrate your Forms to J2EE or not? This is not an easy question. The answer depends very much on your situation. Oracle recommends upgrading your Forms to the web and the latest Forms release, migrate self-service parts of your application to J2EE and start new developments with the J2EE stack as well. In this way you fully protect your investments and take advantage of the features of the new technology stack.

We intentionally used the phrase "reuse Oracle Designer metadata" when describing the purpose of the JDG. Why didn't we write "migrate Oracle Designer metadata?" You can certainly migrate Oracle Forms applications from Oracle Designer with JHeadstart, but there is more to such a migration then just running the JDG and JAG. The questions you should ask yourself when making a decision to migrate Oracle Forms to J2EE include for example:

- Are the users familiar with web browser applications and able to adjust to the new front end?
- Do I need to migrate an existing application to J2EE?
- Does the application require high interactivity? If so, can we provide a user interface experience using JSF components that meets the high interactivity requirements?
- Are your infrastructure and development teams ready for the J2EE technology stack?
- Is the application large, mission critical, or time-sensitive?

Chapter 1 of "Oracle JDeveloper 10g for Forms & PL/SQL Developers : A Guide to Web Development with Oracle ADF" discusses these questions. In this section we discuss two additional questions you need to answer before migrating forms to J2EE:

- Is the application suitable for a partial migration?
- How do I select a migration tool?

Note: It is good to keep in mind that migration from Oracle Forms to J2EE might not be needed. It is also good to keep in mind that, should you decide to migrate existing applications from Oracle Forms to ADF web applications, you need to consider if it would be better to start from scratch using JDeveloper or JDeveloper with JHeadstart. For example, starting from scratch is a good idea when the functionality of the existing Oracle Forms application is out-of-date and does not match the current requirements.

Is the Application Suitable for a Partial Migration?

Larger Oracle Forms applications can often be divided in functional subsystems that have their own user groups. Each user group might use the system differently and have different backgrounds. Subsequently, the questions about familiarity with web browsers and high interactivity requirements need to be answered for each user group. The user groups might also have different satisfaction levels with the functionality of the current system.

In addition, web-based, self-service intranet and Internet applications are becoming more and more popular in organizations because the applications can save a

considerable amount in administrative overhead. This trend leads to a shift of functionality traditionally performed by back-office, data-entry staff (who have frequent use and need high interactivity) to all employees of an organization (who might have infrequent use and are satisfied with lower interactivity) or, for Internet applications, even to customers and suppliers of the organization.

If the trend just mentioned applies to your organization, we recommend starting with a partial migration of your application. This migration has a number of benefits, that together greatly reduce the risks associated with forms migration, for example:

- **You can start on a small scale** by migrating a small subsystem that is not mission-critical that serves end users who are accustomed to web applications.
- **You can evaluate the migration tool** you have selected. Does it meet your expectations?
- **You can collect metrics on the effort** involved in migrating a number of forms. You can extrapolate these numbers to make more reliable estimates of the costs involved in migrating other, bigger subsystems.

How Do I Select a Migration Tool?

The Oracle Forms migration market is expected to grow significantly in the coming years. So, not surprisingly, more and more companies have offerings in this space.

Note: An up-to-date list of validated products can be found at http://www.oracle.com/technology/products/forms/htdocs/Oracle_Forms_Migration_Partners.htm.

Oracle helps you in selecting a company by validating the products that help with migrating forms to J2EE and ADF.

Oracle has validated a number of partner products as being viable for its customers to use when migrating Oracle Forms applications. Although it is beyond the scope of this white paper to directly compare product features, we will provide some considerations you can use when making the decision about a product.

Main Technical Considerations

Apart from non-technical criteria like price, support, documentation, and references, we see two main technical aspects to take into account when making this choice:

- The amount of functionality that is migrated automatically.
- The architecture of the migrated application.

Without comparing JHeadstart to other commercial products, we can mention the functionality JHeadstart provides against the functionality that another tool might

provide. Once you have identified a short list of products to consider, you will then be able to plug in specifics about those products.

Migrating Oracle Forms' PL/SQL Code

With JHeadstart, the runtime architecture of the migrated application is identical to a hand-built ADF-JSF application. However, by design, JHeadstart does not migrate the PL/SQL logic that is coded in the forms. PL/SQL logic should be migrated manually using the following process:

1. Evaluate the type of logic in each trigger and procedure.
2. Determine in which tier or tiers (database, business service, Model, Controller, or View) the functionality should be implemented.
3. Implement the functionality using best practices that apply to the technology used for that tier.

How Much Oracle Forms Code is Converted?

That is, will the resulting application have the same appearance as the original Oracle Forms application, and will it retain the same business logic and code structure as the original. Some products basically rebuilt Oracle Forms in Java; for example, a Java method is available in a library for each Oracle Forms built in procedure. The appearance remains the same because other products may use a Swing-based Java applet to convert the Oracle Forms user interface, which allows them to support typical Oracle Forms features like enter-query mode.

Advantages of JHeadstart for Migration

JHeadstart offers the following advantages:

- JHeadstart-migrated applications are as easy to maintain as applications built from scratch using ADF and JSF.
- Developers can reuse the same skills required for building ADF-JSF applications to maintain and extend JHeadstart-migrated applications.
- JHeadstart-migrated applications can easily be enriched with additional features supported by the JHeadstart Application Generator. This only requires changes to the migrated application definition.
- Developing applications with JHeadstart is similar to developing applications with Oracle Designer and Oracle Forms

Advantages of Other Products for Migration

Other products may offer the following advantages over JHeadstart:

- Migrating applications is fast and low-risk, because close to 100% of the functionality is migrated.

- Applications are relatively easy to understand and maintain for Oracle Forms developers because the structure is similar to Oracle Forms. Oracle Forms developers only need to learn the Java language syntax as well as how to work with the migration tool.
- End users do not need retraining because the appearance is the same as the original form.

Disadvantages of JHeadstart for Migration

Consider the following disadvantages of using JHeadstart for Forms migration:

- A JHeadstart migration can be time-consuming, depending on the amount and complexity of PL/SQL logic in the forms.
- JHeadstart requires Oracle Designer metadata to perform the migration of Oracle Forms applications, and this might require reverse engineering of hand-built forms into Oracle Designer, prior to the migration.
- Functionality and appearance of the application will most likely be different. Therefore, users will need to become accustomed to the new interface, although the ADF JSF JSP file using ADF Faces that it generates contains much more user-friendly features than many applications on the Web now.

Disadvantages of Other Products for Migration

Other products may have other disadvantages as follows:

- They may not use the JSF standard, which allows the application to be run in a browser without the Java runtime. Instead they may use a proprietary Swing-based applet that requires the JVM to run in the browser.
- A Java developer without a background in Oracle Forms may require training in understanding the runtime architecture. This is necessary for development as well as for maintenance.
- If your organization builds other J2EE web applications from scratch, for example using JDeveloper, ADF and JHeadstart, then these applications will have a different architecture than the migrated application. This means that you must invest in two separate skill sets to maintain both types of J2EE web applications.

Your choice will depend on how you value the various pros and cons of each migration product.

LICENSING, SUPPORT, AND OTHER RESOURCES

As mentioned, JHeadstart is a separate product and, as such, it must be purchased separately. As an Oracle Consulting product, it is not yet handled by Oracle Support Services or Metalink. However, you can use the JHeadstart Discussion Forum on OTN to ask questions, raise issues, and log enhancement requests. The JHeadstart Product Center on OTN contains all information related to JHeadstart. You can view and/or download the following:

- **An evaluation copy** of the JHeadstart software. This copy contains all functionality and can be used for pilot projects to evaluate JHeadstart.
- **Comprehensive end-to-end tutorial,** which showcases the main JHeadstart features.
- **Extensive documentation** consisting of the Oracle JHeadstart for ADF Developers Guide, Installation Guide, Release Notes, Migration Guide, Runtime Javadoc, and Designer Generator Reference.
- **Viewlets** that demonstrate the usage of the JHeadstart Application Generator and the JHeadstart Designer Generator.
- **Frequently Asked Questions** including information about the price and purchase details.

The JHeadstart Product Center can be found at http://www.oracle.com/technology/consulting/9iservices/jheadstart.html(or by searching OTN for "jheadstart product center").

Additional information sources follow:

- **JHeadstart Discussion Forum** at forums.oracle.com/forums/forum.jspa?forumID=38
- **JHeadstart Weblog** at www.orablogs.com/jheadstart/

Related resources

For ADF related resources take a look at the following sites:

- OTN JDeveloper Product Center http://www.oracle.com/technology/products/jdev/index.html
- OTN mini-site: J2EE for Oracle Designer and Forms developer http://www.oracle.com/technology/products/jdev/collateral/4gl/formsdesignerj2ee.html
- The new Oracle ADF Developer's Guide for Forms/4GL developers by Steve Muench http://www.oracle.com/technology/documentation/jdev/b25947_01/index.html

Oracle JHeadstart Overview
June 2006
Author: Steven Davelaar
Reviewers: Peter Koletzke, Sandra Muller, Peter Ebell, Ton van Kooten

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com