ORACLE JHEADSTART 10g for ADF

(RELEASE 10.1.2.1)

DEVELOPER'S GUIDE

JULY, 2005

ORACLE®

JHeadstart Developer's Guide

Copyright © 2005, Oracle Corporation

All rights reserved.

Contributors: Steven Davelaar, Peter Ebell, Sigrid Gylseth, Ton van Kooten, Sandra Muller, Jaco Verheul

The programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be licensee's responsibility to take all appropriate fail-safe, back up, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle disclaims liability for any damages caused by such use of the Programs.

This software/documentation contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited.

If this software/documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

> **Restricted Rights Legend** Programs delivered subject to the DOD FAR Supplement are 'commercial computer software' and use, duplication and disclosure of the Programs shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are 'restricted computer software' and use, duplication and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-14, Rights in Data --General, including Alternate III (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free.

ORACLE, Designer/2000, Developer/2000, SQL*Plus, SQL*Loader, SQL*Net, CASE*Method, ORACLE Parallel Server, PL/SQL, Pro*C, SQL*Module are registered trademarks of Oracle Corporation, Redwood City, California.

CDM Advantage, PJM Advantage, Oracle Cooperative Applications, Oracle Financials, Oracle Alert, Oracle Manufacturing, Oracle Inventory, Oracle Bills of Material, Oracle Engineering, Oracle Capacity, Oracle Commissions, Oracle Master Scheduling, Oracle MRP, Oracle Work in Process, Oracle General Ledger, Oracle Assets, Oracle Order Entry, Oracle Cost Management, Oracle Payables, Oracle Receivables, Oracle Personnel, Oracle Payroll, Oracle Purchasing, Oracle Quality, Oracle Sales and Marketing, Oracle Service, and Application Object Library are trademarks of Oracle Corporation, Redwood City, California.

Microsoft and MS-DOS are registered trademarks and Windows, Word for Windows, Powerpoint, Excel, and Microsoft Project are trademarks of Microsoft Corporation. Visio is a trademark of Shapeware Corporation. Project Workbench and Project Bridge Modeler are registered trademarks of Applied Business Technology. All other company or product names mentioned are used for identification purposes only and may be trademarks of their respective owners.

CONTENTS

CHAPTER

1 Overview

Java and J2EE are here to stay. Large software companies as well as a growing segment of the world's development community have embraced this open standard. The promise of such an open standard is tremendous. Component and service based development can become a reality. Application development in the Java/J2EE world will consist of assembling components from a combination of standard, commercially available components and custom components. Web services are available everywhere and can be wired together to support your business processes. This approach means faster development time, better quality and maintainability, and portability across a range of enterprise platforms. The bottom line benefits are increased programmer productivity, more efficient use of computing resources, and greater return on an organization's technology investments.

This nirvana is still not reached yet. Developing transactional applications on the J2EE platform is not a straightforward task. 4GL developers would even argue that moving to Java and J2EE is a tremendous step back in productivity, and they are probably right. What only takes minutes in a 4GL environment can easily take a day or more in the Java world.

So what do we need in order to be as productive in the Java world as in the 4GL world?

First, we need an Integrated Development Environment (IDE) that allows us to work productively. Oracle offers one of the best, if not the best, Java IDE around. *Oracle JDeveloper 10g* provides a comprehensive set of integrated tools that support the complete development lifecycle, from source control, modeling, and coding through debugging, testing, profiling, and deploying. JDeveloper simplifies J2EE development by providing wizards, editors, visual design tools, and deployment tools to create high quality, standard J2EE components including applets, JavaBeans, JavaServer Pages (JSP), servlets, and Enterprise JavaBeans (EJB). JDeveloper also provides a public Addin API to extend and customize the development environment and seamlessly integrate it with external products.

Secondly, we need an application server that is J2EE certified to test and deploy our application. Again, Oracle has one of the best, if not the best Application Server around -- *Oracle Application Server 10g*. Although you can use a range of other IDE's in conjunction

with Oracle Application Server 10*g*, as Oracle, we prefer of course the use of Oracle JDeveloper. The reverse is also true. You can use JDeveloper together with other application servers, but again, as Oracle, we prefer of course that you use the Oracle Application Server. With the introduction of the Oracle BPEL manager as part of the Oracle Application Server it is now possible to easily string web services together to support your business processes. If you have not done so already, this is certainly an area that you should investigate.

Okay -- we have a great IDE, the open standard J2EE platform and one of the best application servers, so developing J2EE applications should be a piece of cake now! Not necessarily so! In spite of all the wizards and the modular platform there are still a lot of things we need to arrange before we can build a transactional application.

How do we communicate with the database? How do we implement a number of best practices in the J2EE world, better known as J2EE Design Patterns and how do we design a good user interface? There is still a lot to do.

To help out in these areas, you can make use of frameworks. Frameworks take care of the low-level plumbing and hide the complexity of developing J2EE applications so that developers do not need to be aware of the underlying complex architecture. Furthermore, a good framework is proven, time tested and improved by going through numerous iterations and enhancement phases. Basically, a good framework has done and will do the dirty work for you, allowing you to keep your hands clean. Therefore frameworks make it easier to get a J2EE application completed in a shorter time frame.

Oracle JDeveloper comes with such a framework called the Oracle Application Development Framework (ADF). The next section gives a short introduction of Oracle ADF and explains how it fits in with what is called the Model-View-Controller Architecture.

The last section introduces JHeadstart; Oracle Consulting's offering that enables **rapid, component-based development** of J2EE applications, standing on the shoulders of Oracle ADF.

Using Frameworks

J2EE specifies a multi tier architecture that distinguishes between server side presentation logic (Web Tier) and server side business logic (EJB Tier).

For server-side presentation logic J2EE supports deployment of dynamic content, with both Java servlets API and JavaServer Pages (JSP) technology. The Java Servlets API enables developers to easily implement server-side behaviors that take full advantage of the power of the rich Java API.

For server side business logic J2EE offers EJB technology, that gives developers the ability to model the full range of objects useful in the enterprise by defining two distinct types of EJB components: Session Beans and Entity Beans. Session Beans represent behaviors associated with client sessions -- for example, a user purchase transaction on an e-commerce site. Entity Beans represent collections of data –-- such as rows in a relational database -- and encapsulate operations on the data they represent. Entity Beans are intended to be persistent, surviving as long as the data they're associated with remains viable.

J2EE extends the power and portability of EJB components by defining an infrastructure that includes standard clients and service APIs for their use.

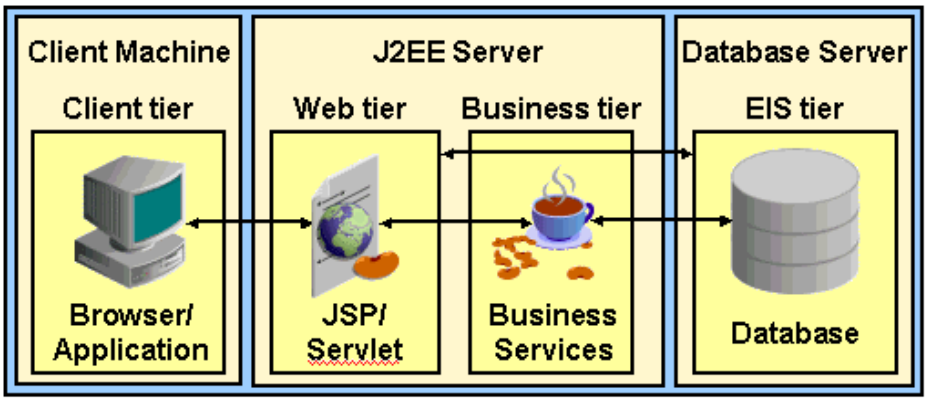


Figure 1-1 J2EE Architecture

Model-View-Controller Architecture

Best practices that have been collected while building J2EE applications have been documented in so-called design patterns. An important architectural pattern is the Model-View-Controller (MVC) pattern.

The MVC architecture can be mapped to multi-tiered J2EE applications as follows:

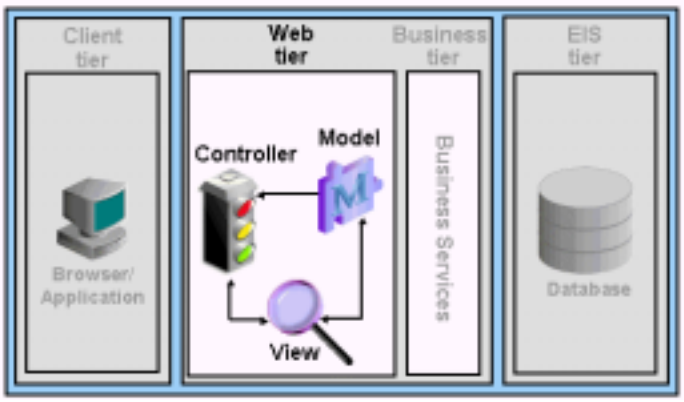


Figure 1-2 Mapping MVC Architecture to the J2EE Platform

All enterprise data and the business logic to process the data can be represented in the MODEL. The VIEW can access the data through the model and decide on how to present them to the client. The VIEW must ensure that the presentation changes as and when the MODEL changes. The CONTROLLER can interact with the view and convert the client actions into actions that are understood and performed by the MODEL. The CONTROLLER also decides on the next view to be presented depending on the last client action and results of the corresponding MODEL action(s).

Applying the above logic to a sample application, you build the application as follows:

- The business logic of the application is represented by EJBs that form the MODEL of the MVC architecture. The MODEL responds to requests from the CONTROLLER or VIEW to access / modify the data it represents.
- The various screens of the application form the VIEW of the MVC architecture. The VIEW updates itself when the MODEL changes.
- The CONTROLLER of the application is a set of objects that receive the user actions, convert them into requests understood by the model, and decide on the next screen to be displayed once the model completes the processing request.

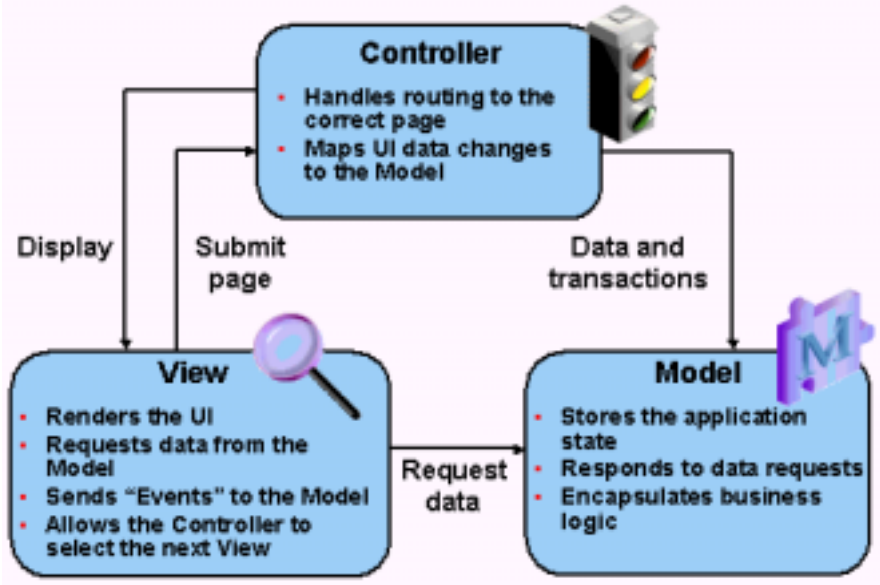


Figure 1-3 Model View Controller Architecture

Key Benefits of MVC Architecture

Key benefits of an MVC Architecture include:

- Support for Multiple Client Devices: you can have different view renderers (phone, PDA, etc.) without changing the controller or model implementation
- Allows you to replace an MVC component without re-writing the whole application
- Prevents vendor lock in
- Allows for choosing the best-of-breed solution for each MVC component

Web Link: For more information on the MVC architecture refer to Sun's book *Designing Enterprise Applications with the J2EE Platform, Second Edition by* Inderjeet Singh, Beth Stearns, Mark Johnson, and the Enterprise Team

http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/app-arch/app-arch2.html#1105854

Oracle Application Development Framework (ADF)

The Oracle Application Development Framework of Oracle JDeveloper 10*g* offers a complete framework to develop J2EE applications. Since it supports multiple technologies you have the choice to use the components that best fit your situation.

It comes with extended design time facilities. By using simple drag-and-drop of the model components you can build page by page in a highly productive manner. For Struts a very useful page flow modeler is included where you can draw the logic of your controller structure. The business services can be developed with several types of wizards (based on UML models), and several types of editors.

Altogether Oracle ADF provides a first class J2EE framework that couples high development productivity with the flexibility to choose the components that fit your situation best.

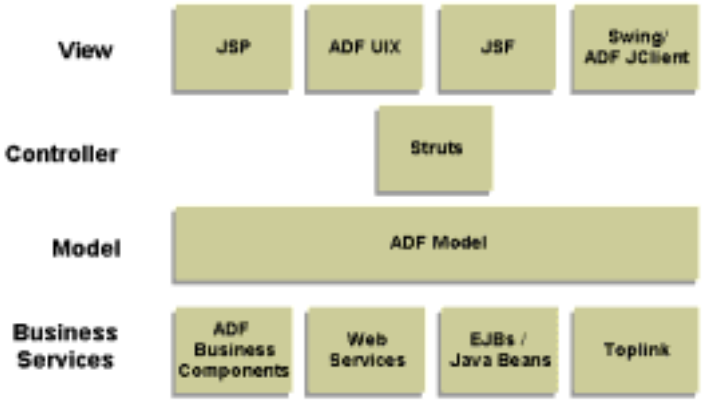


Figure 1-4 Oracle ADF Architecture Implementations

Web Link: For more information on ADF please refer to the JDeveloper Product Center on OTN: http://www.oracle.com/technology/products/jdev/index.html

Business Services – ADF Business Components (ADF BC)

ADF Business Components (ADF BC, formerly knows as BC4J) provides a productive framework for implementation of the Business Services. By using simple wizards ADF BC implements optimized database interaction, business logic encapsulation, J2EE deployment, and performance and scalability.

Key features of ADF Business Components include:

- UML Modeling

  Using Oracle JDeveloper's UML modeling tools, you can model business components in several ways. You can model and generate business components directly from database tables; you can complete your component model then bind the components to database tables; or you can generate database tables from an ADF BC model.

- Object-Relational Mapping

  To develop a J2EE application you need a set of objects that map to tables and object types in your database. Using ADF BC you can quickly generate this set of objects. As well as providing object-relational mapping, the generated components enforce your primary and foreign key constraints, thus maintaining the relationships between your data.

- Business Logic Encapsulation

  ADF BC makes it easy to define your validation and business rules in the middle tier. This way your business rules are enforced in any client that accesses the middle tier. It also allows you to validate your data before it is posted to the database, thus decreasing network traffic between the middle tier and the database.

- J2EE Deployment

  You can deploy your ADF BC application in any of the standard J2EE deployment models: J2EE Web Module, EJB Session Bean, or local deployment as a professional client.

Web Link: For more information on ADF Business Components refer to OTN. http://otn.oracle.com/products/jdev/info/techwp20/wp.html For more detailed information about ADF BC refer to the JDeveloper online help (http://helponline.oracle.com/jdeveloper/help/).

Model

The model component of ADF is an abstraction layer that wraps the business services. It performs the following tasks:

- Handles data events from the Controller
- Feeds data to the View

The big advantage of a model layer is that it abstracts the implementation technology (ADF BC, TopLink, EJB's, Web Services, JavaBeans), of the business services. A page

developer (view and controller) does not need to know this since he is only using the data binding and data controls that are available in the model. These data bindings and controls are available in a drag-and-drop mode within JDeveloper.

Controller – Apache Struts

Apache Struts is an open source controller framework for building web applications. Struts can be described as follows (the text below is taken from the Struts website):

The core of the Struts framework is a flexible control layer based on standard technologies like Java Servlets, JavaBeans, ResourceBundles, and Extensible Markup Language (XML), as well as various Jakarta Commons packages. Struts encourages application architectures based on the Model 2 approach, a variation of the classic Model-View-Controller (MVC) design paradigm.

Struts provides its own Controller component and integrates with other technologies to provide the Model and the View. For the Model, Struts can interact with any standard data access technology, including ADF Model. For the View, Struts works well with ADF UIX, JavaServer Pages, including JSTL, as well as XSLT, and other presentation systems.

Key features include:

- Declarative, XML-based implementation of Controller

  The sequence of application functions (called actions in Struts terms) that need to be executed in response to a user action, the page/process flow, is defined declaratively in a so-called Struts-Config XML file.

- Powerful JSP Tag libraries

  Struts provides a rich set of JSP tag libraries. The Tiles tag library that provides a powerful page templating mechanism is also part of the Struts distribution since version 1.1.

- Extensible and Configurable

  Struts is designed with extensibility in mind. It is easy to add to or override standard Struts functionality.

Web Link: For more information on Apache Struts go to the official Struts homepage: http://jakarta.apache.org/struts/

View – ADF User Interface XML (ADF UIX)

UIX is a set of technologies that constitute a framework for building web applications. The main focus of UIX is the user presentation layer of an application (View). UIX is designed to create applications with page-based navigation, such as an online human resources application, rather than full-featured applications requiring advanced interaction, such as an integrated development environment (IDE).

UIX includes a large set of UI components that implement the Oracle browser-based look-and-feel (BLAF), dynamic image generation, dynamic CSS generation, an XML language for page layout, and a servlet-based application framework.

Oracle Browser Look and Feel (BLAF)

Oracle's UI lab has developed the **Browser Look and Feel** (BLAF) User Interface (UI) Guidelines. This is a set of very detailed standards to help designers, developers and product teams build HTML applications. The purpose of the guidelines is to provide a consistent look and feel, consistent interaction model, and ultimately a consistent user experience across the entire suite of Oracle E-Business products.

Since BLAF has such a wide range of influence, its guidelines have been well tested in the usability lab. Oracle's UI Lab has conducted many usability walk-throughs of HTML applications that follow BLAF, and had extremely positive results. Oracle's UI Lab will continue to test prototypes and live applications, constantly gathering new data to reinforce and add enhancements to BLAF.

Figure 1-5 Examples of components that are described in Oracle's Browser Look and Feel and implemented by UIX

Web Link: UIX implements the Oracle Browser Look and Feel (BLAF). For more information on BLAF please go to OTN (http://www.oracle.com/technology/tech/blaf/index.html)

Changing the UIX Look and Feel

Recently, UIX added support for a second look and feel: the **Minimal Look And Feel**. The Minimal Look And Feel has a somewhat different focus from the Browser Look And Feel. Whereas the purpose of BLAF is to implement Oracle's UI standards, the goal of MLAF is to provide an alternate look and feel that generates "minimal" content. Generating smaller content has several advantages, not the least of which is less network overhead. A less obvious benefit, but perhaps just as important, is that smaller/simpler content lends itself better to content customization. The Minimal Look And Feel can serve as a base for other "custom" look and feel implementations.

With ADF UIX you can apply a certain look or design to a component without adding code to the actual application during design-time. The new "design" will be applied when the component is used at runtime freeing the developer from even thinking about applying templates to the application. If a change is made to a particular component it will show immediately at runtime. No work for the application developer.

So how can we make this possible? The ADF UIX Components are based on a fundamental split:

- UINodes - provide a logical representation of components
- Renderers - provide a physical (e.g. HTML) implementation of component.

This split allows a page description to be defined independently of any markup, or even markup language. This allows us to write one component that can handle multiple agents e.g. different browsers, mobile devices. This also allows us to write new renderers - skins - for different agents separately from the actual development of our applications.

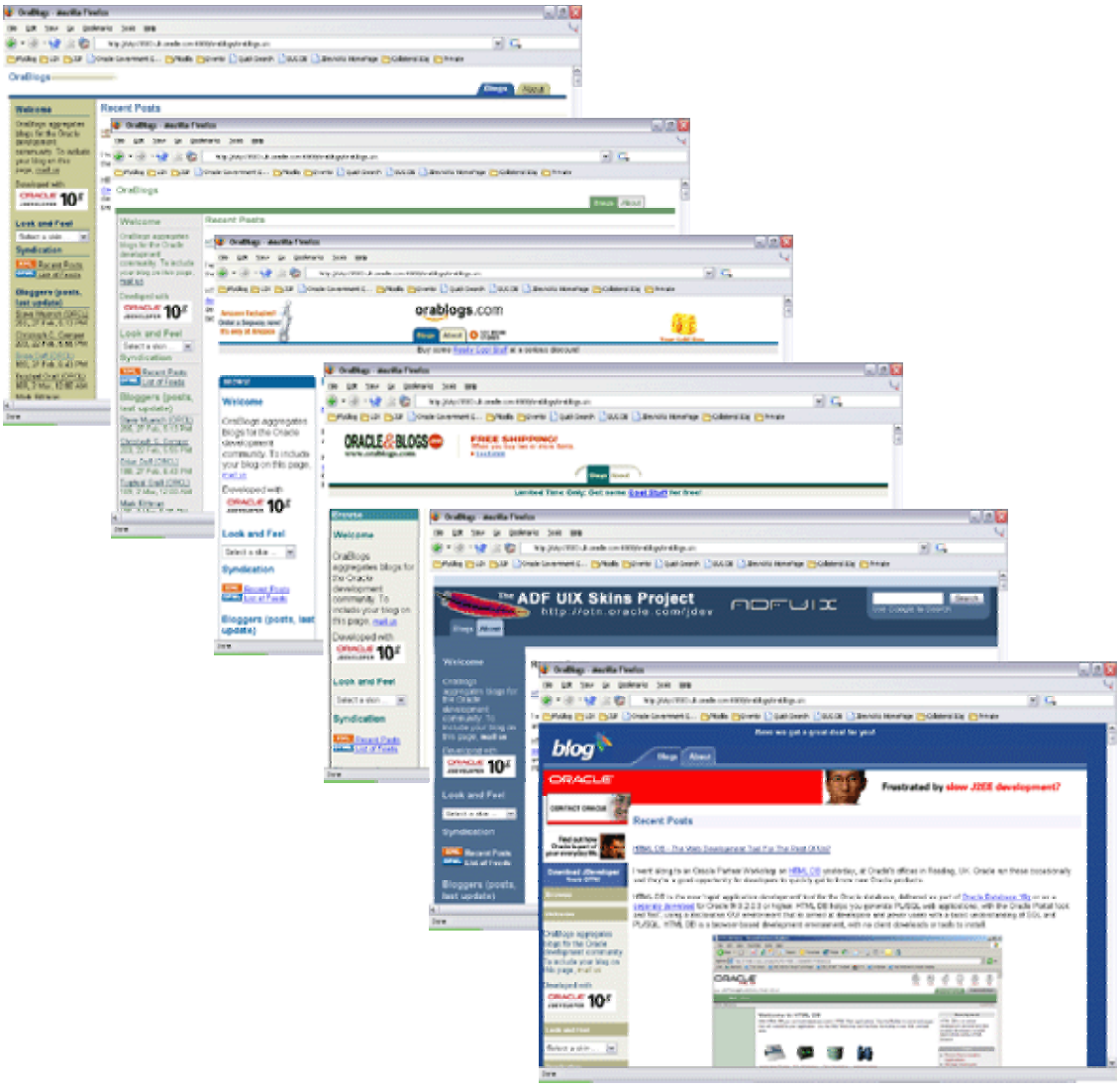


Figure 1-6 Examples of UIX skins

Web Link: Introduction UIX skins:
http://www.orablogs.com/jjacobi/archives/2004_03.html

Web Link: How To create a Look and Feel for ADF UIX:
http://www.oracle.com/technology/products/jdev/howtos/10g/adf_uix_laf_ht/index.html

Key Benefits of UIX

There are many reasons to use UIX:

- UIX provides an **open, flexible framework** for development. You can choose among the different UIX features for different development needs. For instance, you can use UIX Components for rendering pages, or you can use your own HTML or Java Server Pages (JSP) for rendering while still taking advantage of the remaining features of UIX. Additionally, you can use whatever back-end data technologies best suit your needs.
- UIX is **platform independent** because it is implemented in the Java programming language and other portable web technologies.
- UIX **supports a wide range of client agents**, with more to come. For instance, UIX adjusts its presentation for various browsers and locales. It also supports rendering for mobile devices.
- Applications written to the UIX technology stack maintain a **consistent appearance**. The UIX rendering features implement high-level user interface controls, which are consistently rendered across your application (and the applications of others using UIX).
- UIX applications may be **customized at multiple levels**. You can change many aspects of the application independently, including page layout, styles, and imaging, and creating your own look & feel. The environment makes simple customizations easy, and more complicated customizations possible.
- If you choose, much of your UIX development can be **declarative**, using uiXML, an XML language for creating UIX pages and managing application flow. UIX can derive its page layouts, styles, and many other features from uiXML documents, with no programming or compiling involved.
- The UIX architecture has been designed with **localization and internationalization support** in mind. Its rendering technologies automatically adjust for the target client's locale, and the framework is built to help separate localizable content from the user interface.
- **High performance** has been designed into the framework, such as the caching and reuse of shared resources.

By providing these features, UIX helps to reduce the amount of work needed to get an application running, tested, and customized.

Web Link: For more information on UIX please refer to the UIX Developer's Guide (http://helponline.oracle.com/jdeveloper/help/)

View – Java Server Pages (JSP)

Java Server Pages (JSP) is a server side technology that controls the content of web pages through the use of servlets, small programs that are defined in the web page and run on the server to modify the page before it's downloaded to the user who requested it.

Key benefits of JSP include:

- JSP is easy to learn and allows developers to quickly produce dynamic web pages.
- JSP integrates nicely with JavaBeans technology
- JSP's are platform and browser independent

JSP is by far the most popular technology to create web pages within J2EE applications.

Choosing and Combining ADF components

JHeadstart uses these components to form a complete application:

- ADF BC implements the Business Services it contains business objects and business logic (business logic can also be implemented in the database using CDM RuleFrame),
- ADF Model implements the model abstraction layer it contains data bindings and data controls;
- UIX or JSP implements the View; it handles your pages and a consistent look and feel.
- Apache Struts implements the Controller; it handles user requests and communication with the business objects.

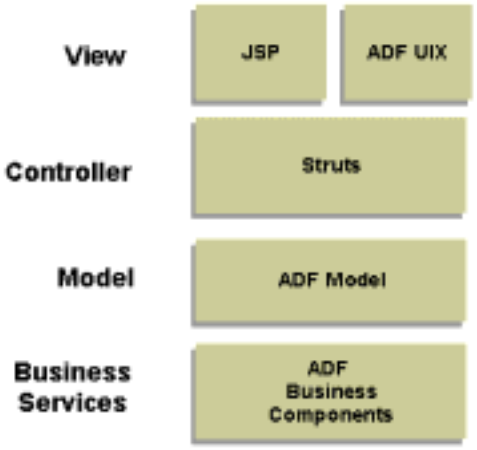


Figure 1-7 JHeadstart use of ADF Components and Frameworks

Note that all components are standards compliant, and run on any J2EE compliant application server, not just Oracle Application Server.

Choosing the View

For the view you can choose between UIX and JSP.

UIX gives you a lot of functionality out of the box, and since it is fully customizable it can be adapted to your standard look and feel by using so called skins. The next version of UIX (called ADF Faces) will support the Java Server Faces standard (previews are already available from on OTN). In other words, if you want to be as productive as possible and get in line with the new J2EE standards, UIX should be the view implementation of choice.

Benefiting from the rich component set that UIX offers, JHeadstart allows you to use the following features if you choose UIX as the view technology (for more information about these features see Chapter 3, JHeadstart Application Generator):

1. Generation of Trees and Shuttles
2. Use List of Values for validation
3. Multi-select List of Values
4. Detail-disclosure in table layout

If you want to use an industry-standard View technology and one that every Java developer masters to some degree, JSP is the better choice.

What is Oracle JHeadstart 10*g*?

JHeadstart is a development toolkit, fully integrated with JDeveloper, that enables rapid component based development of J2EE applications. It provides you with 4GL-like productivity without jeopardizing the flexibility and openness of the J2EE architecture. It is fully based on Oracle ADF.

JHeadstart consists of three main components:

- JHeadstart Runtime

  The JHeadstart runtime contains reusable components that extend Oracle ADF. These reusable components implement Oracle ADF best practices that were developed during custom development projects of Oracle Consulting.

- JHeadstart Application Generator (JAG)

  Apart from the runtime components, JHeadstart provides significant design-time support. The JHeadstart Application Generator (JAG) is a powerful generator that automates the development of the Controller (struts config file) , View (UIX or JSP files), and Model components (ADF data controls and data bindings). The JAG is driven by XML meta-data that you create using JDeveloper (plug-in) wizards and JHeadstart property editors, providing you with a declarative, 4GL-like experience in building J2EE applications. To help you to get started with the meta data, JHeadstart generates a first cut of the meta data based on your ADF Business Components, which can be retrieved from a UML class model or database tables.

- JHeadstart Designer Generator (JDG)

  In addition, JHeadstart offers you a migration path from the Oracle Forms/Designer world to the Java/J2EE world. Using the JHeadstart Designer Generator, the meta-data in the Oracle Designer Repository are transformed into the XML meta-data format required by the JHeadstart Application Generator. If you have a manually built Forms application, you can use the Designer Design Capture facilities to load this information into the repository. You can then run the JDG, followed by the JAG to migrate your forms application to a professional J2EE application that implements the MVC design pattern!

Development Steps

While developing applications with JHeadstart you will normally go through the following steps:

1. **Create ADF Business components** either from existing database tables or a UML Class Diagram using the JDeveloper ADF BC wizards.
2. **Generate your first cut application meta** data by letting JHeadstart do that for you.
3. **Generate your application** with the JHeadstart Application Generator, test it and show it to the end user community when necessary.
4. **Refine the meta data** using the JHeadstart property editors.
5. **Generate your application** using the JAG and repeat steps 4 and 5 as many times as you think is necessary. Since refining and generating only takes seconds

or minutes you can iterate with the end user next so he can indicate what needs to be changed.

6. **Create complex pages using JDeveloper ADF design time interface** (drag and drop, model or code interface)

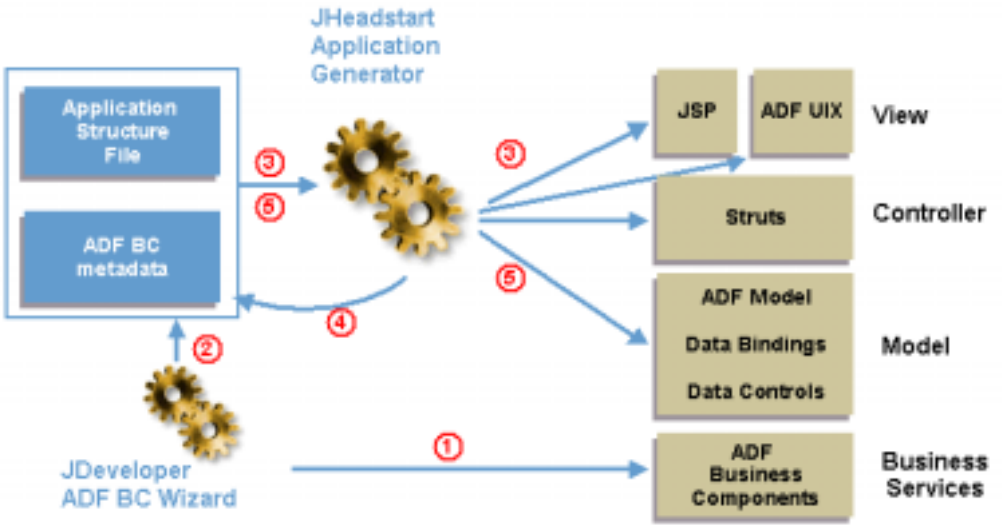


Figure 1-8 Roadmap for developing JHeadstart Applications

For a more detailed list of steps please refer to Chapter 3, JHeadstart Application Generator.

The meta data can be refined (step 4) with simple property editors that allow you to specify your application. Most of the properties have dropdown- lists that enable you to specify your meta data by simple mouse-clicks. All properties are clearly explained but most of them speak for themselves. With a click on the generation button you can generate the complete application straight from this editor.

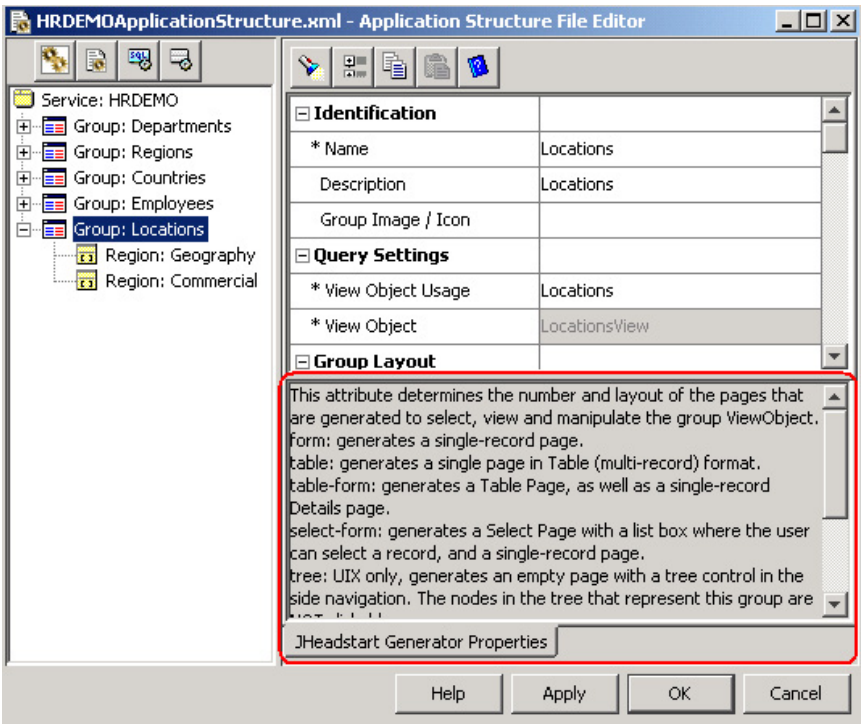


Figure 1-9 Easy-to-use property editor to refine meta data

Advantages of JHeadstart

Using JHeadstart you will have the following advantages:

- Unprecedented J2EE developer productivity
- Makes transition from any 4GL language to J2EE much easier
- Only limited knowledge required to get started
- Consistent application because of the generation process
- You immediately benefit from the best practices of Oracle Consulting., since runtime components have been added to JHeadstart on the bases of Oracle Consulting's worldwide experience in ADF projects.
- No java is generated only web pages and the Controller XML file. This Struts file configures and wires the different run time components together. This dramatically simplifies customization and maintenance of the generated application.
- Fully based on ADF, which allows you to use the powerful design time interface of ADF
- ADF UIX allows you to easily set up your company's look and feel
- Standards based; applications run on every J2EE certified application server
- Makes migration to new standards (for instance Java Server Faces (JSF) easier because new versions of the generator that support these new standards can fully leverage existing meta data and just generate the application again.
- Can also be used to develop prototypes that support a business case for a new development project.

For the ones that have a Designer/Forms background the following extra advantages apply:

- Makes the step to J2EE easier because way of working is very similar to Designer/Forms. ADF is very similar to Forms Builder and JHeadstart is similar to the Designer Forms Generator.
- Protects current investments because of reuse of metadata in de Oracle Designer Repository: It allows you to generate back-end applications using Oracle Forms and self-service or Internet applications (html) with JHeadstart based on the same metadata!
- Seamlessly integrated with CDM RuleFrame, all your business rules in RuleFrame are also validated through the J2EE application.
- Helps you to migrate Forms to J2EE when desired.

What does JHeadstart Generate?

JHeadstart generates the following pages and page structures on the basis of simple meta data.

Lay-out styles:

Form, Select-Form, Table, Table-Form, Tree, Tree-Form, Master-Detail page with detail in same or separate page (unlimited nesting of parent-child relations), Regions, Shuttle with multi-select functionality. All Form pages come with previous, next and first and last buttons. You can specify the number of columns on a form page.

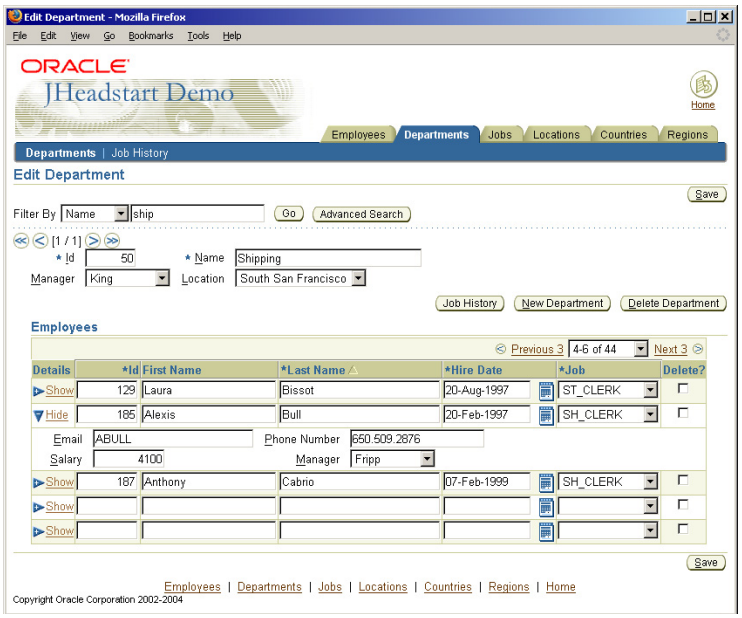


Figure 1-10 Example of a Master-Detail page with detailed disclosure

Table layout extra's

Sortable column headers, hide details in table page, detail disclosure (UIX only) that discloses the details in the table when you click on the disclosure button, choosing number of records displayed which results in a poplist and next and previous set buttons

Query

Quick Search, which gives a poplist with all the fields you can search on; Advanced Search, which gives a Search Form where you can combine search criteria; Auto Query; Query operators like is, is not, less, and greater. The search results can be displayed in the same or a different page.

Transactional Behavior

For every page you can specify if a single or multiple insert, update and/or delete is allowed. JHeadstart generates table pages that allow you to do multiple inserts, updates and deletes at the same time, which allows for easy recording of data. When you allow multiple inserts in tables you can specify the number of new rows.

Figure 1-11 Example of a Tree-form layout page

User Interface Widgets

You can set: prompts, default display value (including using expression language to define your default), display types like text, checkbox, choice (based on a particular domain), list, editor, date, date time, display only, secret, file upload (results in a field with a browse button to search the file to be uploaded), file download and images.

Lookups

Can be displayed either as a list of values (separate window) or a choice (poplist). The list of values can be used for validation, this enables the user to enter part of the value and if there is only one the value it will be auto completed, if there are more values the list of values will be shown. With the multi-select list of values the end user can choose multiple values at once. Using query bind parameters you can make list of values dynamic, and for instance base a list of values on a value that has been chosen in a previous list of values on the same page.

Customizing the JHeadstart generator templates

JHeadstart comes with generator templates which allow you to change the template and to set tokens so items are generated in a place where you want to have them.

Internationalization

JHeadstart supports that your application is in a different language than English and comes packaged with support for a large number of languages. In addition JHeadstart supports the development of multi language applications.

Figure 1.12 Example of a multi-select component

How to read this Developer's Guide

Determine the phrase that best fits your character and skill level. Are you a

- Quick Starter
- Architect
- Java Guru
- Forms to J2EE mover
- Designer lover
- Or a I-want-to-know-everything kind of person

Choose and read your section.

Quick Starter

Even a quick starter should read something before he starts. Well, you are well on your way, because you have almost finished this chapter. When done, you may proceed with Chapter 2 *Getting Started*. After that you can start generating your J2EE application with the information in Chapter 3 *JHeadstart Application Generator*.

Architect

The Architect should familiarize himself with the Oracle Application Development Framework. More information can be found on the Oracle Technology Network. You can proceed with Chapter 5 about *JHeadstart Extensions to ADF Runtime* that gives you a good idea what JHeadstart adds to the Oracle Application Development Framework. If you are interested in the generation process you should read the first sections of Chapter 3 *JHeadstart Application Generator*.

Java Guru

Java gurus should know everything about J2EE frameworks. This means you should familiarize yourself with Oracle ADF and especially ADF Business Components and ADF UIX. If you didn't work with Struts yet, you are probably not a Java Guru. Then again there is enough information on the Internet to help you get started with Struts. You can proceed with Chapter 5 about *JHeadstart Extensions to ADF Runtime* that gives you a good idea what JHeadstart adds to Oracle ADF.

If you hate generators you should skip chapter 3. If on the other hand you are practical and want to improve your productivity you should proceed with Chapter 2 *Getting Started* and then learn how to drive the *JHeadstart Application Generator* (Chapter 3) in a matter of minutes.

Forms to J2EE mover

You should start with the Statement of Direction "Migrating Forms Applications to J2EE" on OTN and the Oracle Tools Statement of Direction

Web Link: Both statements of direction can be found on OTN.

The statement on Direction of Forms or J2EE can be found here:
http://www.oracle.com/technology/products/forms/htdocs/10g/FormsJavaSOD.html

The Tools statement of direction can be found here:
http://www.oracle.com/technology/products/forms/pdf/10g/ToolsSOD.pdf

If you are still convinced migration to J2EE is something you would like to investigate, continue with Chapter 4 *JHeadstart Designer Generator* and read the *JDG reference* in the appendix. If you want to try it out you should start with Chapter 2 *Getting Started* and also read Chapter 3 if you want to get more out of the migration by using the JHeadstart Application Generator.

Designer lover

If you love to work with Designer and really don't want to change your way of working but are interested to see how you can generate J2EE applications out of Oracle Designer, start with the Chapter 2 *Getting Started* and Chapter 4 *JHeadstart Designer Generator* and start generating out of Designer. Also use the *JDG Reference* in the appendix for more information.

Then if you like the result you should probably go to a Java course or at least have someone in your office that can explain more about Java/J2EE and object oriented programming. You should then familiarize yourself with Oracle ADF. OTN is a good place to start.

I-want-to-know-everything-person

Of course if you want to know everything you should read everything. Reading the developer's guide is much more fun when you actually try out JHeadstart for yourself and discover all the functionality that JHeadstart brings you. We suggest you use a simple database design of about 6 tables (for example the HR schema mentioned in chapter 2), and build a fully-functional application on top of it, as you go through the chapters.

CHAPTER

2

Getting Started

Every time you begin an application development project using JHeadstart, there are a few simple steps you must perform. The first time you do this, we recommend that you try it out on a sample schema, creating your own JHeadstart demo application. Therefore this chapter describes how to create a JHeadstart application based on the HR sample schema (which is also used for the JDeveloper tutorials). When you create your own application, you can apply the same steps to your own application schema.

This chapter discusses the following steps:

This chapter assumes that you have already performed the steps listed in the Installation Guide. The Installation Guide can be found in the root folder of your JHeadstart installation (install.html).

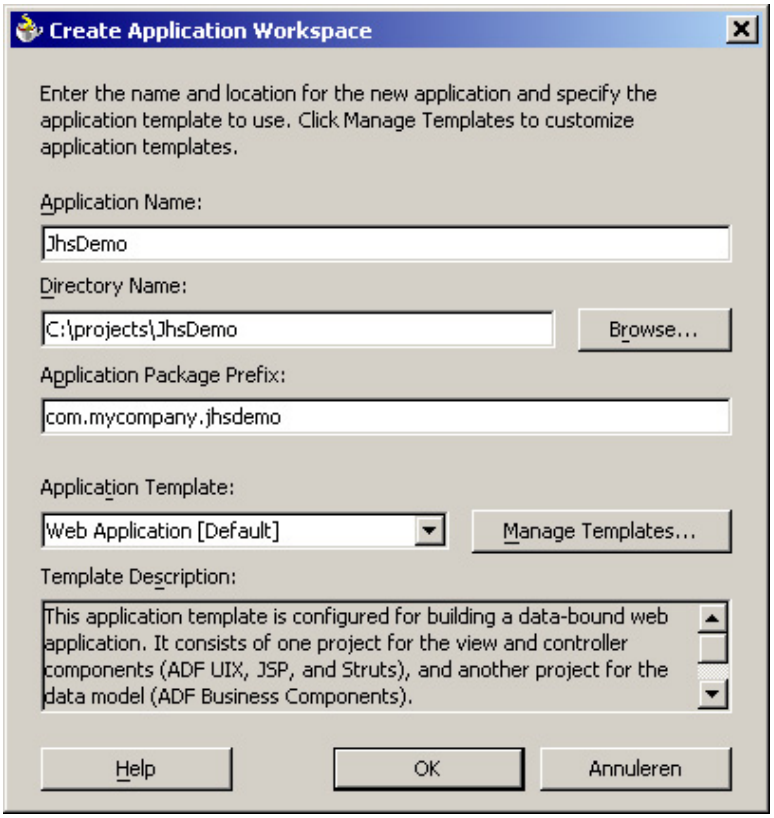
Setting Up the JDeveloper Project Environment

Although you can use JHeadstart in any kind of JDeveloper project, the recommended way is to use an Application Workspace for an ADF Web application, with separate projects for the Model (holding the ADF Business Components and other Services you might want to use in your application) and for the Web Tier (View and Controller).

JDeveloper Workspace and Projects

JDeveloper offers a convenient wizard for setting up an Application Workspace and Projects.

- Create a New Application Workspace (choose menu option File – New – General – Application Workspace).
- Choose a name and directory for the new workspace, and also type in a default package name (for example, `com.mycompany.jhsdemo`).
- In the Application Template field, choose **Web Application [Default]** from the dropdown list.



This will create two projects in your workspace: one called Model and one called ViewController. In the Model project you can set up the ADF Business Components, and in the ViewController project JHeadstart can generate the View and Controller layers of your application.

Setting Up the Business Components Package

If you are using the JHeadstart Designer Generator, read Chapter 4, *JHeadstart Designer Generator*, for information about how to create Business Components and an Application Structure File for your project using the JDG. Then continue with section 'Building Your Application' in this chapter.

If you are not using the JHeadstart Designer Generator, you will need to create ADF Business Components yourself. One way to do that is to create a Class Model in JDeveloper and create your Business Components from there. For this chapter, however, we'll use the JDeveloper wizard for creating Business Components based on existing database tables.

Creating a Database Connection

Create a Database Connection to the schema that contains the database tables of your application.

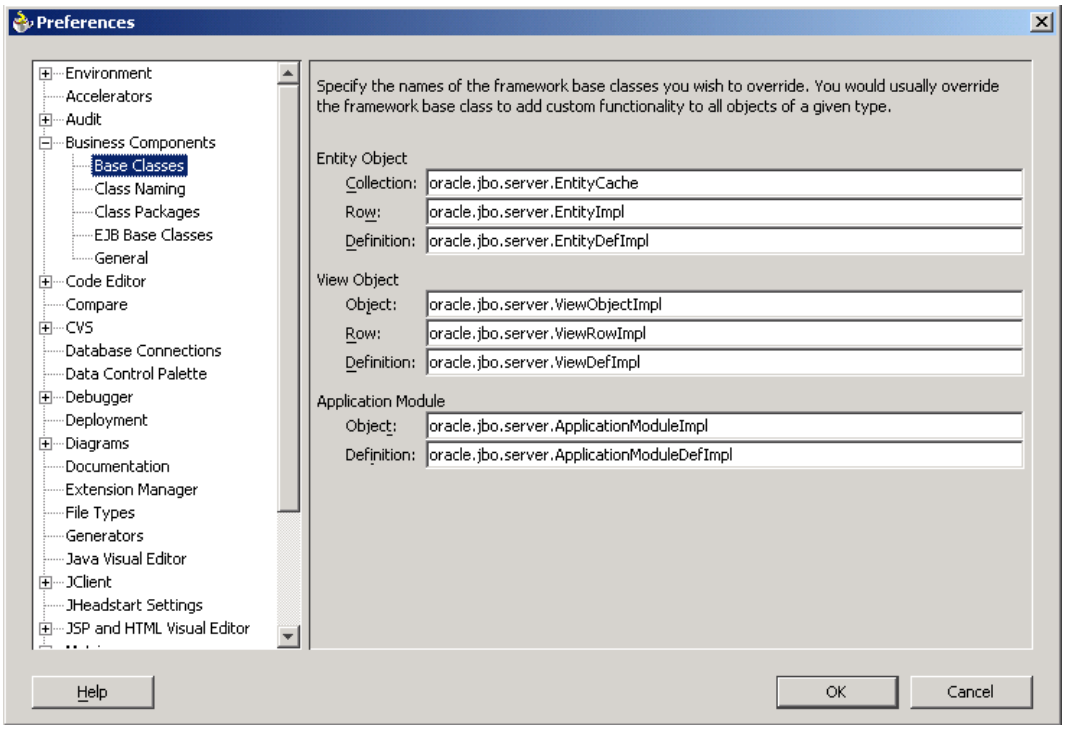
For the JHeadstart demo application you can use the HR (Human Resources) schema that is also used for the JDeveloper tutorials.

Reference: Follow the steps in the JDeveloper tutorial 'Installing the Sample Schemas and Establishing a Database Connection' http://www.oracle.com/technology/obe/obe_as_1012/j2ee/common/obeconnection.htm

Setting Up ADF BC Base Classes

Attention: For the demo application, this section can be skipped entirely. However, for a production application, it is recommended to apply the steps in this section.

Every type of ADF Business Component extends from a Base class. By default, the base classes are set to the standard ADF BC classes defined in oracle.jbo.server package. You can check that under menu option Tools – Preferences.



In the whitepaper 'Business Rules in ADF BC' and in the documentation of the 'ADF Toy Store Demo' you are advised to create your own ADF BC Entity Object super class, that in turn extends the standard ADF BC `oracle.jbo.server.EntityImpl` class.

Suggestion: Quote from Steve Muench (Toy Store Demo): *I recommend always creating a set of ADF framework extension classes, even if you currently have no particular need to. When you later need to address a new feature that affects all components you have created of a given type, you will be super glad that you listened to this advice. I work with customers who setup multiple layers of framework customization classes for their business components. A first layer is a "company wide" set of classes that extend the base components in `oracle.jbo.server.*`. For each application project they work on, they create a project-level set of framework customization classes as well. You can set up your preferred ADF Business Components base classes at the IDE level, under the Tools | Preferences... dialog, on the Business Components > Base Classes panel. If you want to override these global settings for a particular project, you can also visit the Business Components > Base Classes panel on the Project Properties dialog.*

Although you can extend all eight ADF BC base classes, there are two for whom it is highly recommended that you create your own base classes.

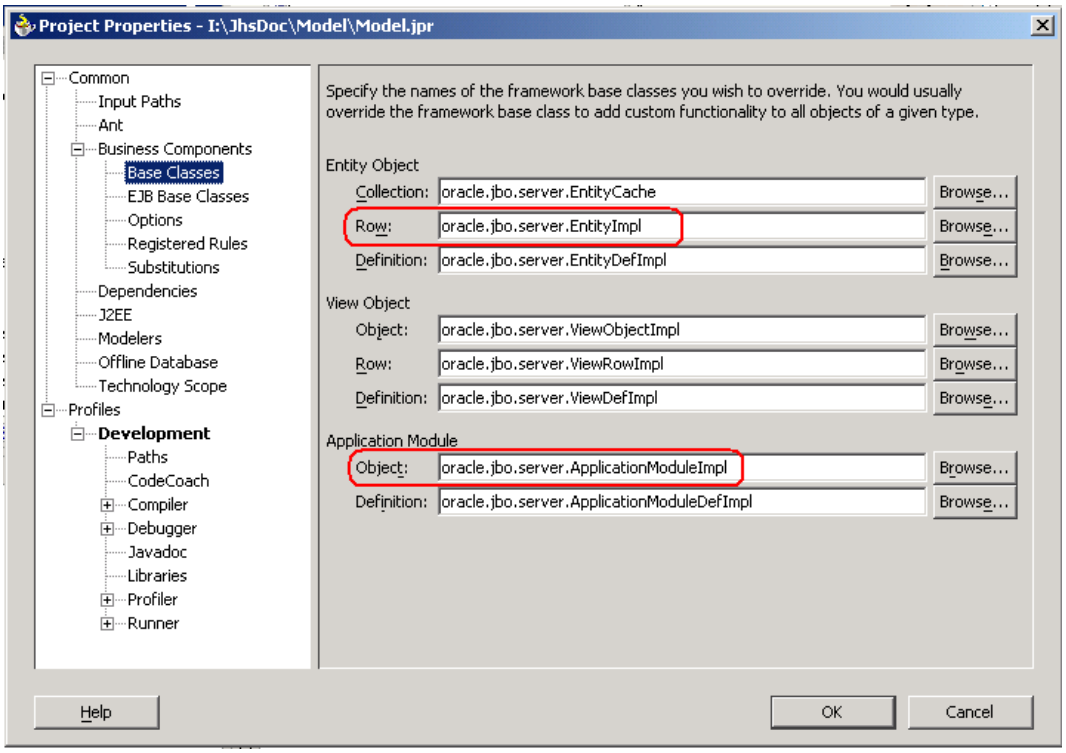
1. The Entity Object Row base class can be used, among other things, to implement default values, sequencing, population of audit columns, and other application wide business rule implementations. Your base class can have any name and be located in any package, but it must extend **oracle.jbo.server.EntityImpl**.

2. The Application Module Object base class can be used to implement functionality that is needed in all Application Modules of your application. Therefore it might not come as a surprise that JHeadstart has already created a subclass of the standard oracle.jbo.server.ApplicationModuleImpl class. (The JHeadstart Application Generator will automatically set up the use of this class.) If you want additional custom functionality for your application modules, this means that you should not extend the standard base class but rather the JHeadstart base class**:** **oracle.jheadstart.model.adfbc.JhsApplicationModuleImpl**.

Attention: When you create an application module class that extends JhsApplicationModuleImpl, you should first generate your application once so JHeadstart will add the JHeadstart Runtime Library to your model project. Otherwise your model project will not compile. See section Building Your Application for information how to do this.

If you decide to create further base classes, you must take care to always extend the standard base class in your own class. Example: <MyViewRowImpl> should extend oracle.jbo.server.ViewRowImpl.

To ensure that the ADF Business Component framework and wizards use your base classes instead of the default ones, you will need to go to the project properties of your BC model project and enter the names of the new base classes:



You are now able to plug in new functionality by changing the base classes. In the ToyStore demo this is used for example for forcing lower/upper case values in attributes.

Reference: For information about enforcing business logic within the ADF BC Business Service, see the whitepaper 'Business Rules in ADF BC', which will be published on Oracle Technology Network (http://www.oracle.com/technology) soon.

This is the successor of the whitepaper BC4J Business Rules in BC4J: http://www.oracle.com/technology/products/jdev/htdocs/bc4j/BusinessRulesInBc4j.pdf

Reference: The ADF Toy Store Demo, including an excellent technical whitepaper by Steve Muench explaining how the demo was built, can be found at http://www.oracle.com/technology/products/jdev/collateral/papers/10g/adftoystore.html.

Creating Default Business Components

Similar to the 'ADF Toy Store Demo', we will set up the ADF Business Components in 3 separate packages. This makes it easier to quickly find the Business Component you are looking for. The packages are:

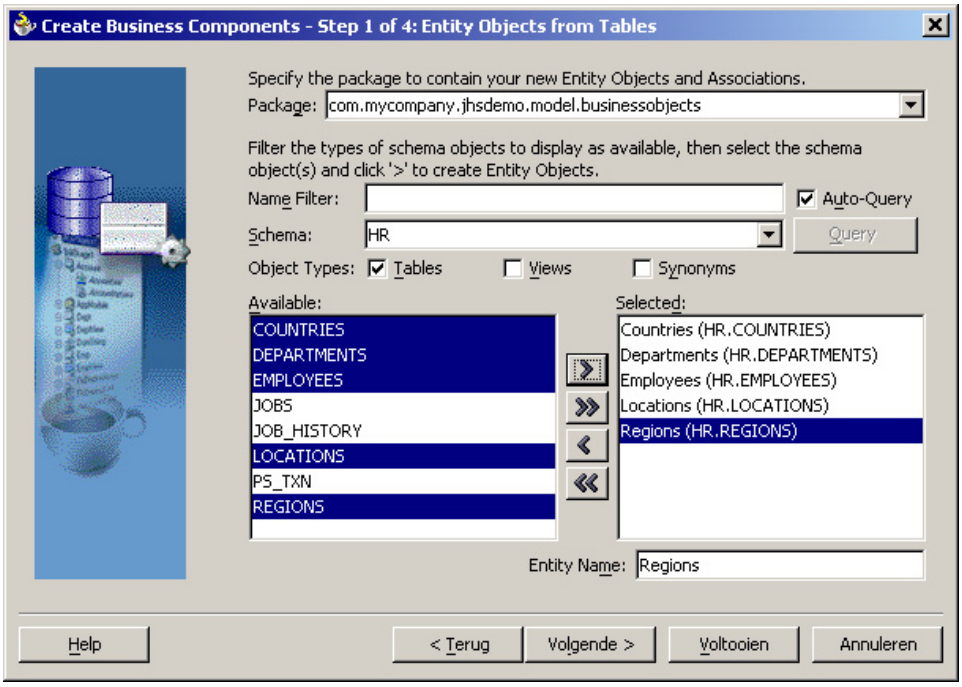
Package Name	Containing ADF Business Components
….model.businessobjects	Entity Objects, Associations and Domains

….model.dataaccess	View Objects and View Links
….model.services	Application Modules

To create your default ADF Business Components you must perform the following steps:

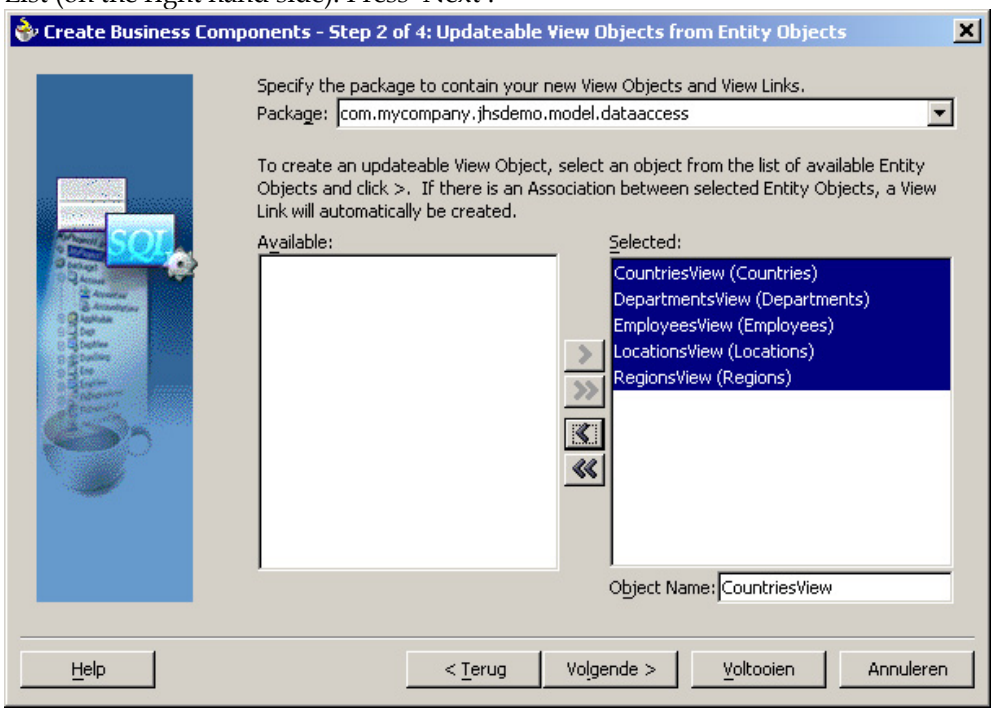
1. Select your Model project, right click and select New...
2. Expand the Business Tier node, select Business Components, and choose Business Components from Tables.
3. Select the connection to the schema that contains the application tables (for example 'HR'), and press OK. The Create Business Components wizard opens.
4. In Step 1 of 4 (Entity Objects from Tables), change the Package to be …model.businessobjects.
5. Select the tables you want to use (in case of the JHeadstart demo, choose COUNTRIES, DEPARTMENTS, EMPLOYEES, LOCATIONS and REGIONS), and press Next.

Attention: In some cases, you will find a PS_TXN table in the HR schema. You should never create an Entity Object for this table. This is an automatically created table to store Application Module state at runtime. More information on this and other technical tables used by ADF Business Components can be found at OTN in 'Overview of Temporary Tables Created By BC4J' http://www.oracle.com/technology/products/jdev/htdocs/bc4j/bc4j_temp_tables.html

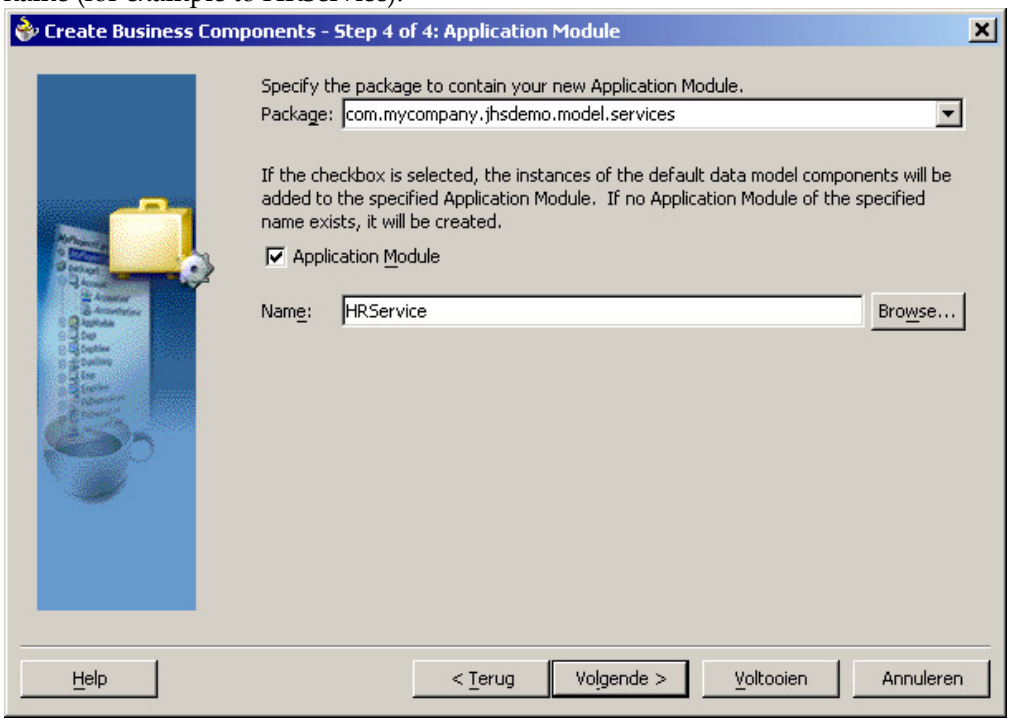


6. In Step 2 of 4 (Updateable View Objects from Entity Objects), specify the Package …model.dataaccess.
7. Ensure that default View Objects are created for all the available Entity Objects (on the left side), by pressing the >> button. Now, all the objects are listed in the Selected

List (on the right hand side). Press 'Next'.



8. Skip Step 3 of 4 (Read-Only View Objects from Tables) for the JHeadstart demo. For your own application, you can of course define any Read-Only View Object you need.
9. In Step 4 of 4 (Application Module), specify the Package …model.services.
10. Ensure that the checkbox to create an Application Module is checked, and change the name (for example to HRService).



11. Click on 'Finish' to create the Business Components.

Adjusting the Data Model of the Application Module

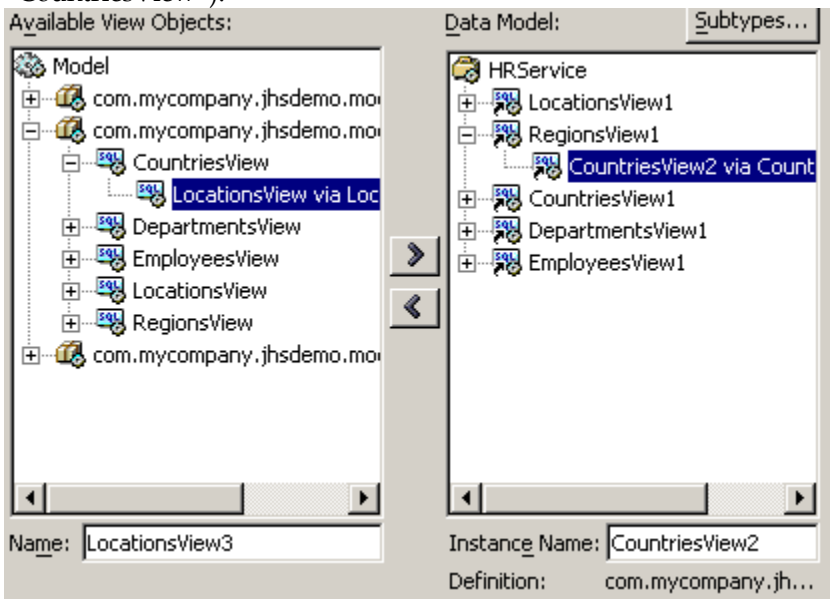
When you created the default Business Components, usages of the View Objects were included in the Data Model of the created Application Module. By default, all View Object Usages are nested one level deep (reflecting all parent-child relations).

Every parent-child-grandchild-etc relation that must be supported in your application must be present in the Application Module's data model. For the JHeadstart demo, we want a nesting of 5 levels deep: Regions – Countries – Locations – Departments – Employees.

To adjust the Data Model of the Application Module, do the following:

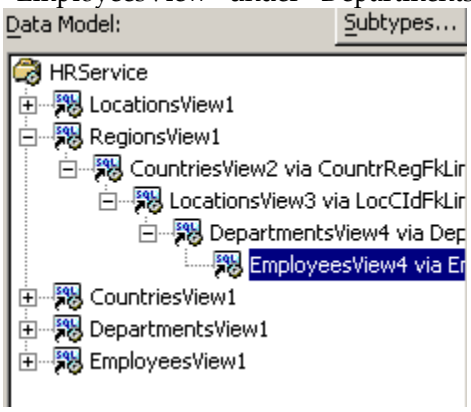
1. Edit the HRService Application Module. You will probably see that the Data Model already contains a View Object Usage "RegionsView1", which has a child usage "CountriesView2".

2. Create a usage for View Object "LocationsView" as a child usage below "CountriesView2". You might need some perseverance here: the user interface of the JDeveloper wizard is not that user-friendly. When adding a detail view, it is important to select the view **as a child view** in the list of Available View Objects (in this example "LocationsView" is selected as a child view of "CountriesView").



Then select the view instance on the right-hand side to which you want to add a child usage (in this example select "CountriesView2" under "RegionsView1"). Then you can add this child view to the Data Model of the Application Module by selecting the button with the '>'.

3. In a similar way, add “DepartmentsView” under “LocationsView3”, and add “EmployeesView” under “DepartmentsView4”. You should end up with this:



4. Click ‘OK’

Suggestion: You might wonder about the numbers at the end of the ViewObject usage names. These usage names are just defaulted that way; you may change them to anything you like. For this demo there is no need to do that, but remember that for a real application it is the View Object Usage names that you will encounter in the Data Control palette. Therefore, when creating ‘real’ applications, it would be advantageous to provide logical names for View Object Usages, for instance “CountriesWithinRegion” instead of the default name “CountriesView2”.

Testing the Application Module

Before continuing with the View and Controller layers of the application, it is a good idea to test the Model (or rather, the Business Service) layer. For just this purpose, JDevelopers offers a built-in Business Components Tester that can be invoked on any Application Module.

Right-click the “HRService” Application Module and choose ‘Test…’ . Check the Database Connection name and click the Connect button. Now the Oracle Business Component Browser opens.

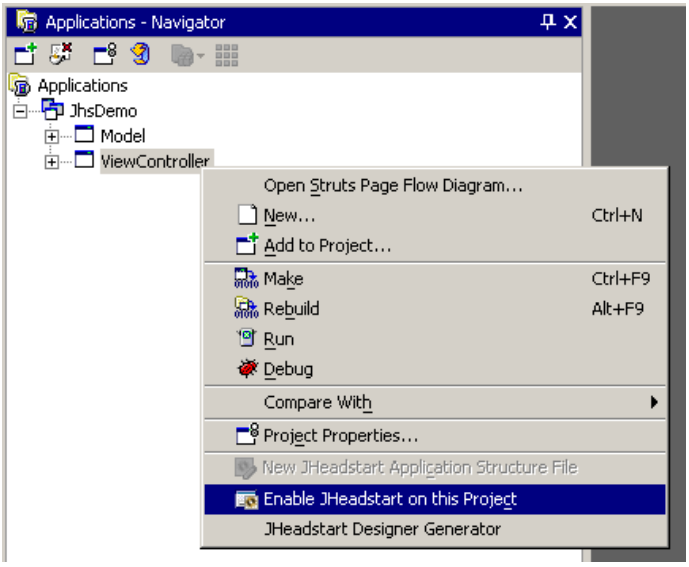
On the left hand side you will see the Data Model of the Application Module. Double click one of the View Object Usages to open a browser for it. On the right hand side you can now browse through the rows, make changes to them, and, using the toolbar, even create and delete rows.

Building Your Application

Now that we have defined our Business Components in our Model project, we can use JHeadstart to generate the other application components in our ViewController project.

Enabling JHeadstart in the ViewController project

Before you can use JHeadstart in a project, you must first "Enable JHeadstart" on it. Locate your cursor on the ViewController project, right-mouse click, and select 'Enable JHeadstart on this Project'.



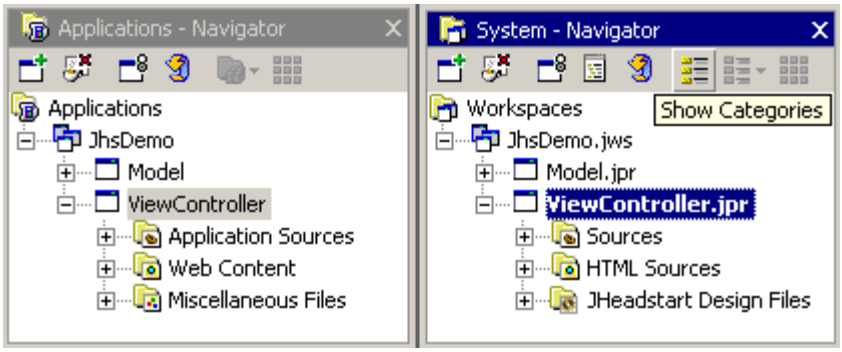
Follow the steps in the wizard.

Attention: Even if you had already started developing your application using ADF, you can still add JHeadstart capabilities by JHeadstart-Enabling the existing ViewController project. See Chapter 3, *JHeadstart Application Generator*, for more information on the "Enable JHeadstart" feature.

Using the System Navigator

JDeveloper 10*g* offers two types of navigators: the default is the Application Navigator, and the alternative is the System Navigator (which is similar to the navigator of JDeveloper 9*i*).

We suggest you use the System Navigator when working with JHeadstart, because then you will be able to group the JHeadstart files under JHeadstart nodes, which makes it much easier to find them.



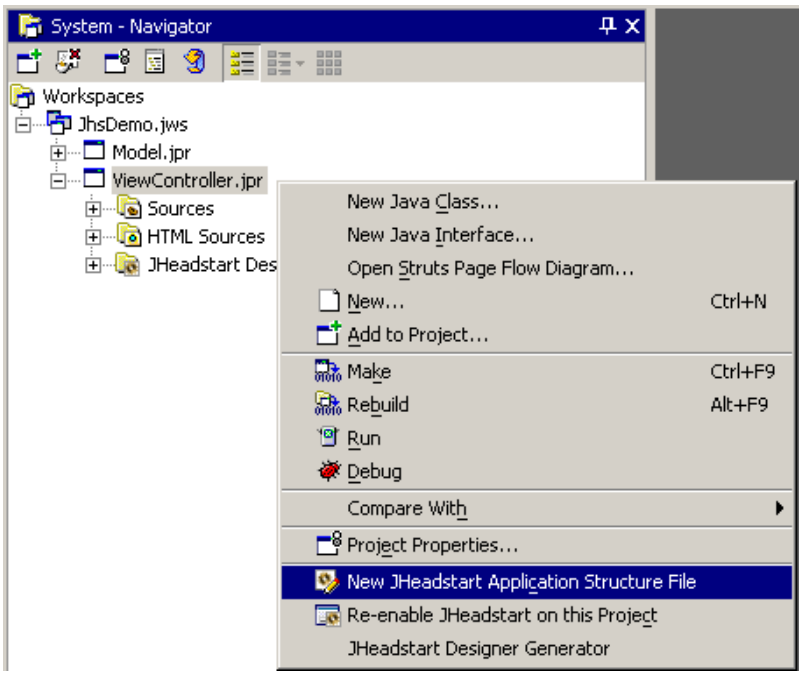
Open the System Navigator by choosing the menu option View – System Navigator. Enable the categories (the grouping nodes) by clicking the Show Categories button or choosing the menu option View – Options – Show Categories (only enabled when the System Navigator is selected).

Creating the Initial Application Structure File

The JHeadstart Application Structure File contains a high-level description of the application you want JHeadstart to generate. If you use the JHeadstart Designer Generator, you do not have to perform this step, as the JHeadstart Designer Generator will generate this file for you.

You can easily create an initial Application Structure File based on the Data Model of an ADF BC Application Module in your workspace.

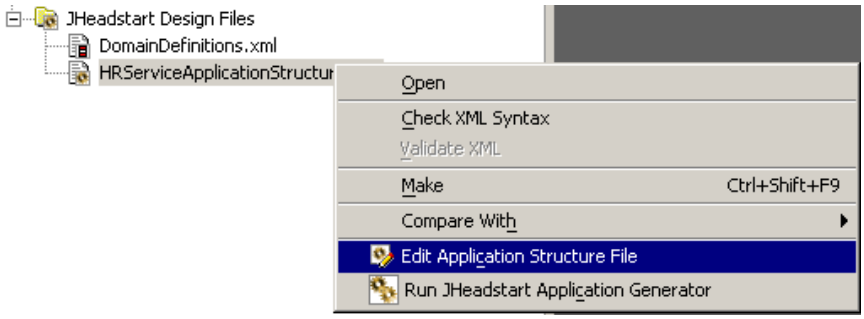
Place the cursor on the ViewController project, right-mouse click and select New JHeadstart Application Structure File.



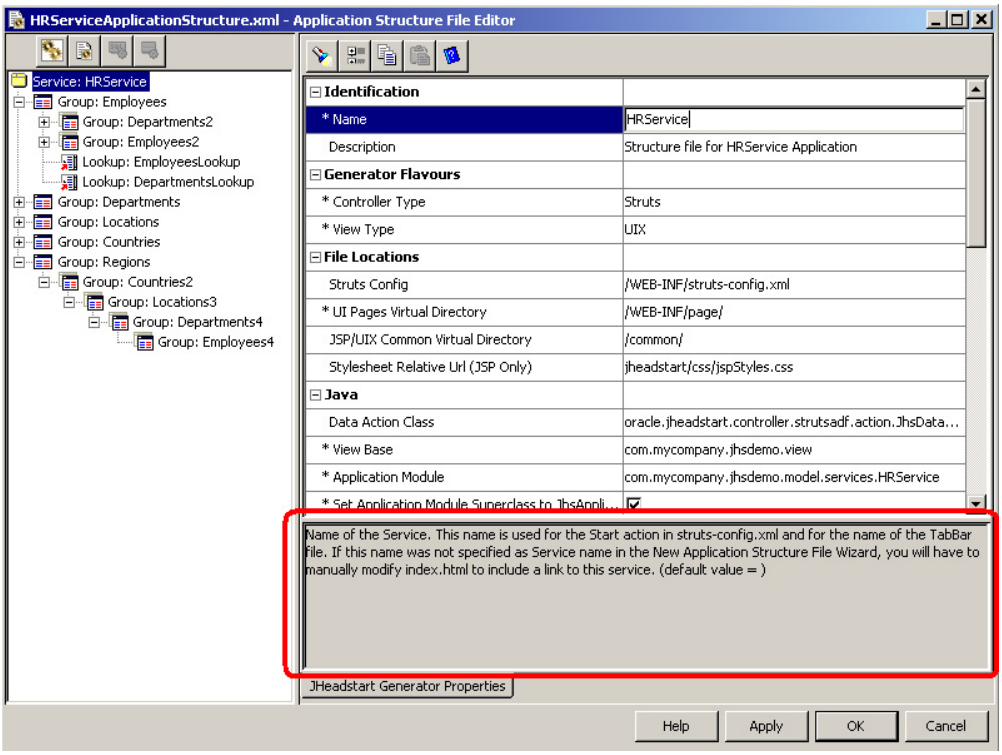
Answer the questions asked by the wizard (you can accept the defaults in most cases) and press Finish. A new Application Structure File will be created, which will appear under the node 'JHeadstart Design Files' in the System Navigator.

Editing the Application Structure File

To edit this file, right-mouse-click the Application Structure file, and select 'Edit Application Structure File'.



Once you have opened the Application Structure File, you must set the properties according to your own needs. For each property, there is a description at the bottom that explains what the property is used for. You can easily expand this area by moving the line above the description:



Note that the New Application Structure File Wizard automatically translated the Data Model of the Application Module to a Group structure under the Service definition (see the left hand side of the Editor).

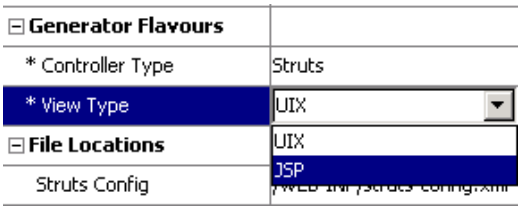
Choose View Layer Technology

JHeadstart is capable of generating two types of View layers:

1. JSP: Java Server Pages.
2. UIX: User Interface XML

In Chapter 1, *Overview*, the relative merits of these technologies were already discussed.

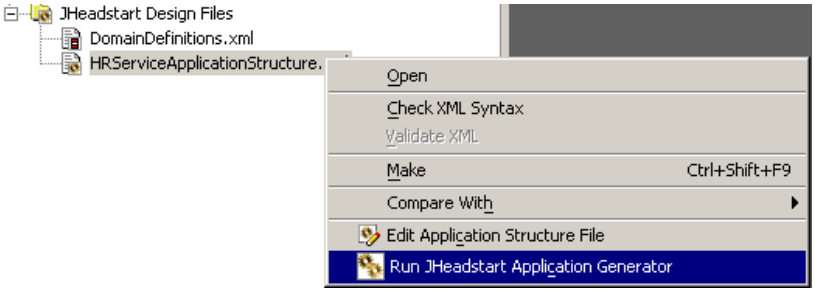
For the demo application, you can select the **View Type** of your choice (JSP or UIX); you can even switch back and forth between runs of the Generator. For the screenshots in this chapter we are going to assume UIX.



Creating a First-Cut Application

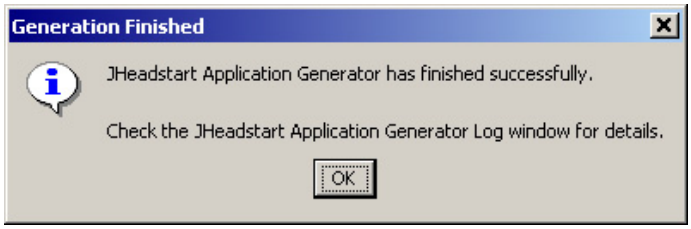
Now we will run the JHeadstart Application Generator (JAG). There are two ways to start JAG:

1. Right click the Application Structure File in the (System) Navigator, and choose Run JHeadstart Application Generator.



2. From within the Application Structure File Editor, click the top left button.

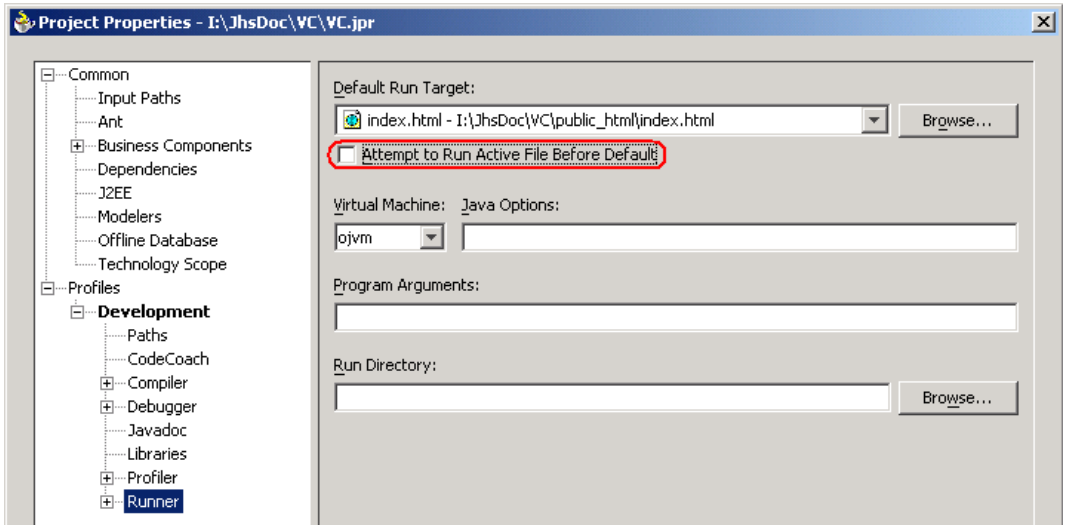
Now JAG will generate the View and Controller layers of the application, including the ADF Model data bindings to the ADF Business Components. When it is finished, you will see this dialog box:



Running the Application

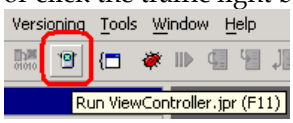
The Enable JHeadstart Wizard has created a file index.html (if it did not exist yet) that can be used to run the generated application. The New Application Structure File Wizard adds a new entry in index.html for each Application Structure File (Service). This index.html file has also been set as default Run Target.

It is recommended to make the default run target be chosen over running the active file. The generated pages should not be run directly. To make it easier to start up the application when you are currently at a page, go to the project properties of your ViewController project and uncheck 'Attempt to Run Active File Before Default'.



Having the default run target (and having unset the checkbox mentioned above) means that you can run the generated Application in one of the following ways:

- From the menu choose Run – Run ViewController.jpr, use the shortcut key F11, or click the traffic light button just below the Versioning menu.

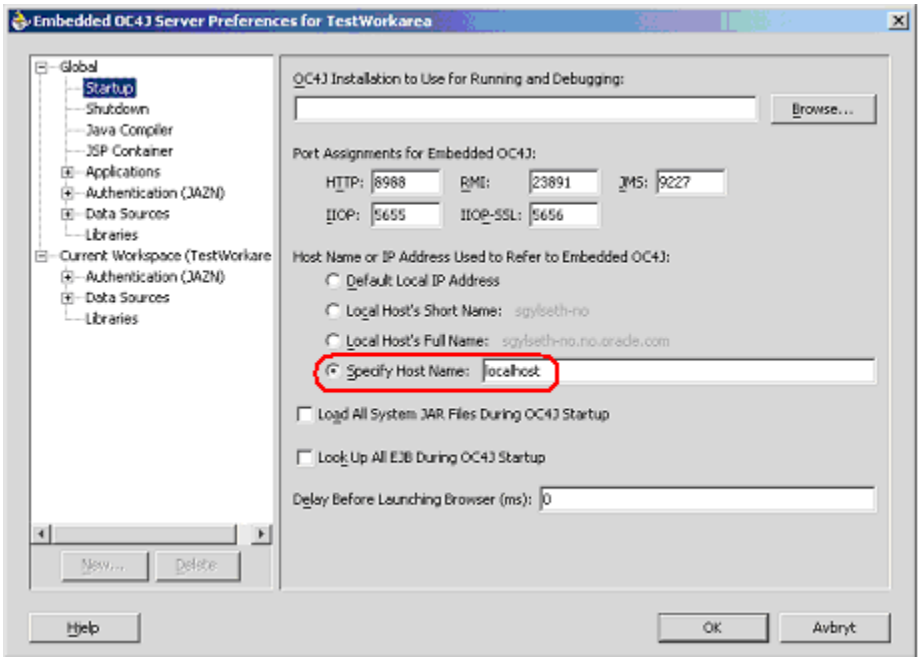


- Right click index.html in the Navigator, and choose Run.

- Right click struts-config.xml in the Navigator, and choose Run. When a list of possible actions appears, choose StartHRService.

- In the Struts Page Flow Diagram, right-click the StartHRService action, and choose Run.

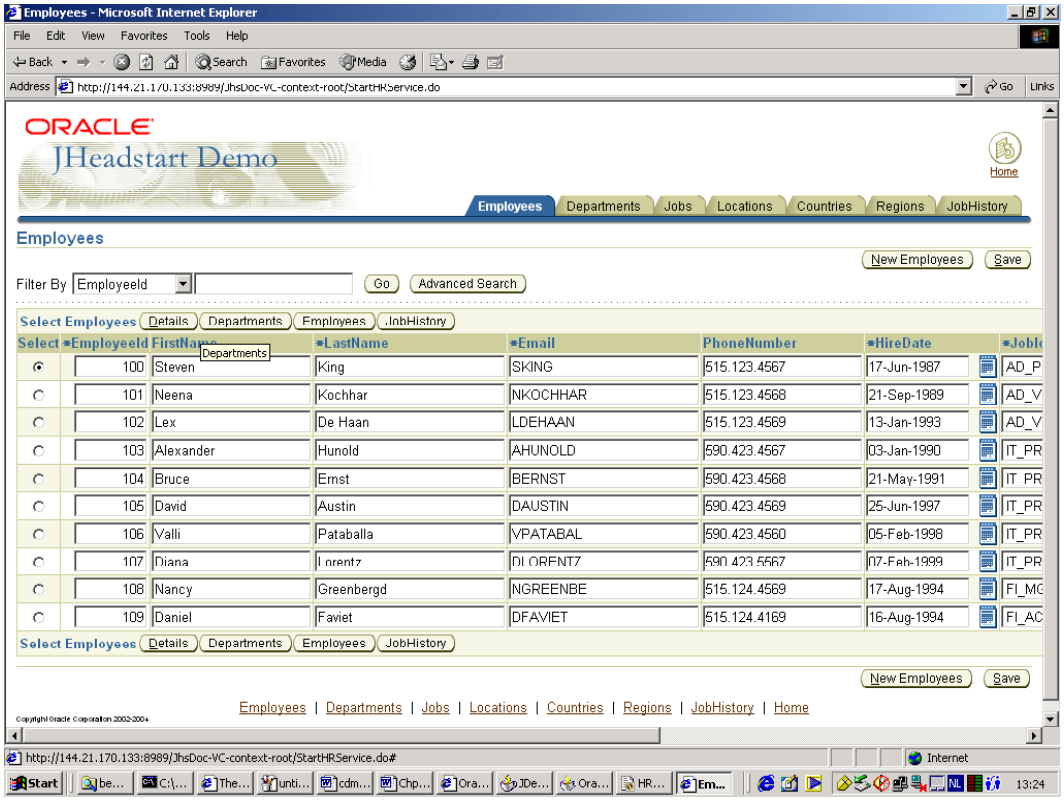
If the application (index page) does not show, and your browser "hangs" or gives a Gateway Timeout, it could be that the proxy settings of your browser don't make an exception for the host name or IP address used by embedded OC4J. Go to the menu option Tools -> Embedded OC4J Server Preferences, and click on Startup below the Global node. Select the radio button "Specify Host Name" and set the value to "localhost". Usually the browser is then able to find the local machine.



Try out the generated application

You will see that JHeadstart has generated a lot of working pages. Have a good look at the generated application, browse through the pages with the tab bar, and try to save some changes.

Your start page should look like this:


Employees - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://144.21.170.133:8989/JhsDoc-VC-context-root/StartHRService.do
Go Links
ORACLE
JHeadstart Demo
Home
Employees Departments Jobs Locations Countries Regions JobHistory
Employees
New Employees Save
Filter By EmployeeId Go Advanced Search
Select Employees Details Departments Employees JobHistory
Select EmployeeId FirstName LastName Email PhoneNumber HireDate JobId
Departments
100 Steven King SKING 515.123.4567 17-Jun-1987 AD_P
101 Neena Kochhar NKOCHHAR 515.123.4568 21-Sep-1989 AD_V
102 Lex De Haan LDEHAAN 515.123.4569 13-Jan-1993 AD_V
103 Alexander Hunold AHUNOLD 590.423.4567 03-Jan-1990 IT_PR
104 Bruce Ernst BERNST 590.423.4568 21-May-1991 IT_PR
105 David Austin DAUSTIN 590.423.4569 25-Jun-1997 IT_PR
106 Valli Pataballa VPATABAL 590.423.4560 05-Feb-1998 IT_PR
107 Diana Lorentz DLORENTZ 590.423.5567 07-Feb-1999 IT_PR
108 Nancy Greenbergd NGREENBE 515.124.4569 17-Aug-1994 FI_MG
109 Daniel Faviet DFAVIET 515.124.4169 16-Aug-1994 FI_AC
Select Employees Details Departments Employees JobHistory
New Employees Save
Employees | Departments | Jobs | Locations | Countries | Regions | JobHistory | Home
Copyright Oracle Corporation 2002-2004
http://144.21.170.133:8989/JhsDoc-VC-context-root/StartHRService.do#
Internet
Start
13:24

Refining the Generated Application

In this section we will make changes in the JHeadstart metadata of our demo application, which will cause the JHeadstart Application Generator to generate the application differently.

Attention: This section describes some features of JHeadstart on a high level. For each of these features, other variants might be possible that are not described here. For a more detailed description refer to Chapter 3, *JHeadstart Application Generator*.

Unless specified, you can apply the refinements to both JSP and UIX as View Type. You can either run JAG and test the application after each refinement (in the same way as described in the previous section), or you can apply all the changes in the meta data in one go, and then generate and run the fully refined application.

Changing a Descriptor Attribute

In many cases, an attribute is needed to show the context on a page. Imagine a master-detail situation where the master and the detail records are on different pages. On the detail screen you need to show information about the master the details belong to.

For each group you can specify the attribute JHeadstart will use to show context information. This attribute is called the Descriptor Attribute.

The New Application Structure File Wizard derives a default Descriptor Attribute for each Group (View Object). The algorithm used is: find the first required String attribute of the View Object, and if none found, use the key attribute. The idea is that if you have a (non-descriptive) Number as Key Attribute, there is usually some other String attribute that is more descriptive, like a Name.

In the case of RegionsView the derived Descriptor Attribute is RegionId (which is a String), while RegionName would be more appropriate. So let's change that.

- Edit the HRServiceApplicationStructure, find the group called Regions, and change its Descriptor Attribute (in the category Labels) from RegionId to RegionName.

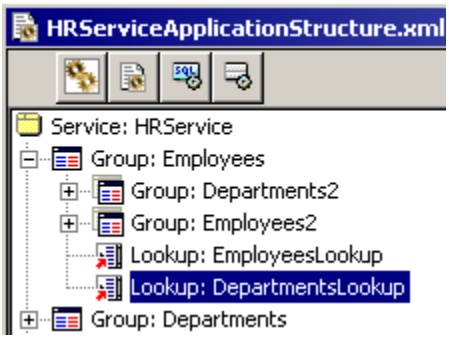
- While we're editing the Labels of that Group anyway, let's also change the Display Title (Singular) from 'Regions' to 'Region'.

You can see the effect in the running application when you click the Regions tab, and then go to the Countries of a certain Region. The Descriptor Attribute is encircled in red in the screen shot below. The Display Title (Singular) is shown as prompt just before the red circle.

Attention: JHeadstart can only have a single attribute as Descriptor Attribute. In case you need to show more attributes for complete context, you have to create an extra attribute for this purpose. See Chapter 3, *JHeadstart Application Generator*, section 'Creating a Logical View Descriptor'.

Generating a List Of Values Window

The New Application Structure Wizard has created Lookups in the Application Structure file for groups that have a parent ViewObject. For example the Employees Group of HRApplicationStructure has a DepartmentsLookup, which allows you to choose the Department of an Employee from a list.



Lookups can have 2 display types: 'choice' (the default, which generates a dropdown list) and 'lov' (a list of values window).

When there are a large number of rows to choose from, the 'lov' type is preferable. It allows the user to perform a search, and show only a limited number of rows at a time.

For the demo application, we will create a List Of Values window for choosing the Department of an Employee.

- Edit the HRApplicationStructure file using the Application Structure File Editor.
- Open the Employees group node and select the DepartmentsLookup.
- Change the Display Type of the lookup from 'choice' to 'lov'.
- If your View Type is UIX, you may also check the property "Use LOV for Validation? (UIX Only)".

In the resulting application the DepartmentId attribute has a flashlight icon (see the screenshots below).

Clicking on the icon produces a List of Values window.

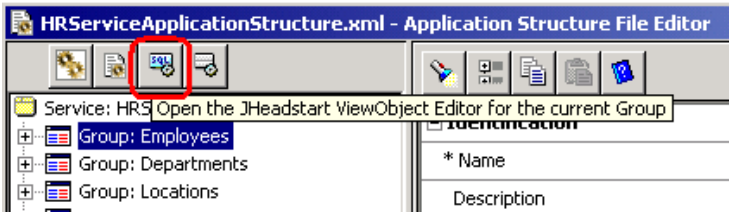
Suggestion: This example causes the Department Id to be shown in the base page. If you prefer to keep the Department Id hidden and show the Department Name instead, see Chapter 3, *JHeadstart Application Generator*, for instructions how to generate that.

Generating Detail-Disclosure

When you have a page with a table layout, by default JHeadstart will put all attributes in separate columns of the table (when **Display in Tables?** = true). When **Display in Tables?** = false, the attribute is not shown in table layout, but only in form layouts.

Let's first ensure that for Employees, only EmployeeId, LastName, FirstName and DepartmentId are included in the table.

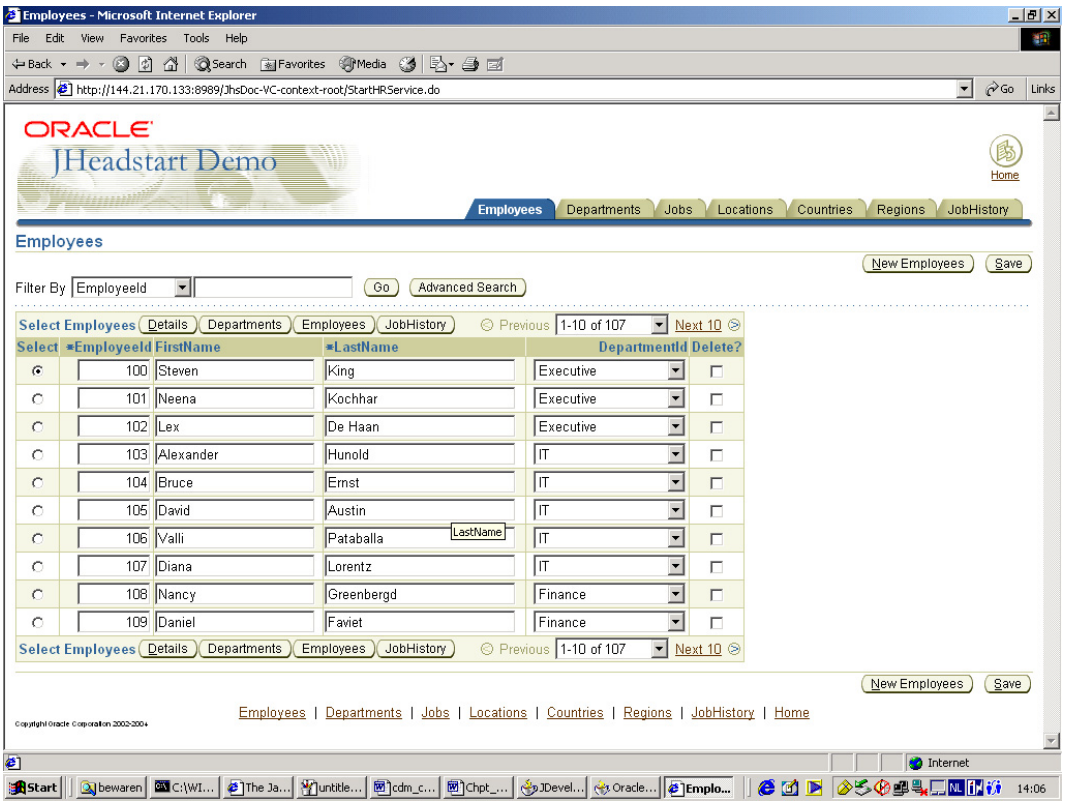
- Edit the HRServiceApplicationStructure file.
- Select the Employees group.
- Click the 3rd button from the left, which will open the JHeadstart ADF BC Property Editor for the relevant ViewObject.



- Multi-select (shift-click) the attributes Email through ManagerId and set 'Display in Tables' to false.

- Click OK and return to the Application Structure File Editor.

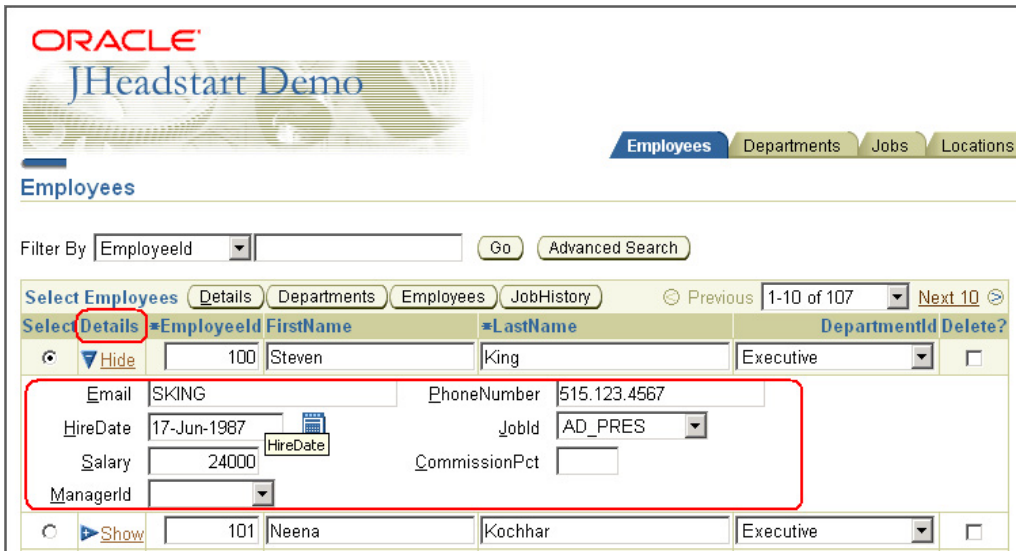
The result will be that the Employees table page will have only a few columns:



When using UIX, you can make the other attributes like Salary editable, while still keeping the table layout. This is done using the Detail-Disclosure feature:

1. Set **Detail Disclosure (UIX only)?** To true for the Employees group.

2. Generate the application. A Details column is added to the table. By clicking on the Show link for a row, the non-table attributes for that row become editable.



Generating a Tree

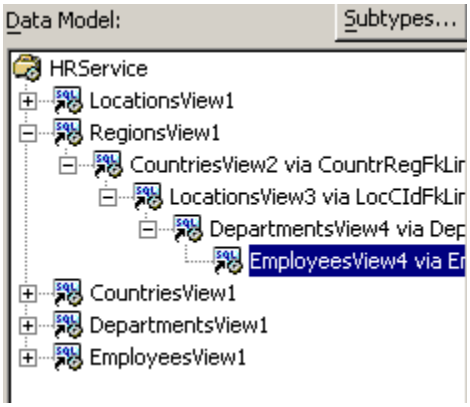
When your view type is UIX, you can use JHeadstart to generate tree controls. A tree control is extremely useful for showing hierarchical structures in your datamodel. This section will explain how to generate a basic tree control with JHeadstart. You will be surprised by the elegancy.

Attention: Again, Chapter 3: *JHeadstart Application Generator*, has more details about generating trees.

In the HR sample schema, a geographical structure is present that can be used in a tree control. We have REGIONS, consisting of multiple COUNTRIES, consisting of multiple LOCATIONS, consisting of multiple DEPARTMENTS, consisting of multiple EMPLOYEES.

This is why we added extra View Instances in section Adjusting the Data Model of the Application Module

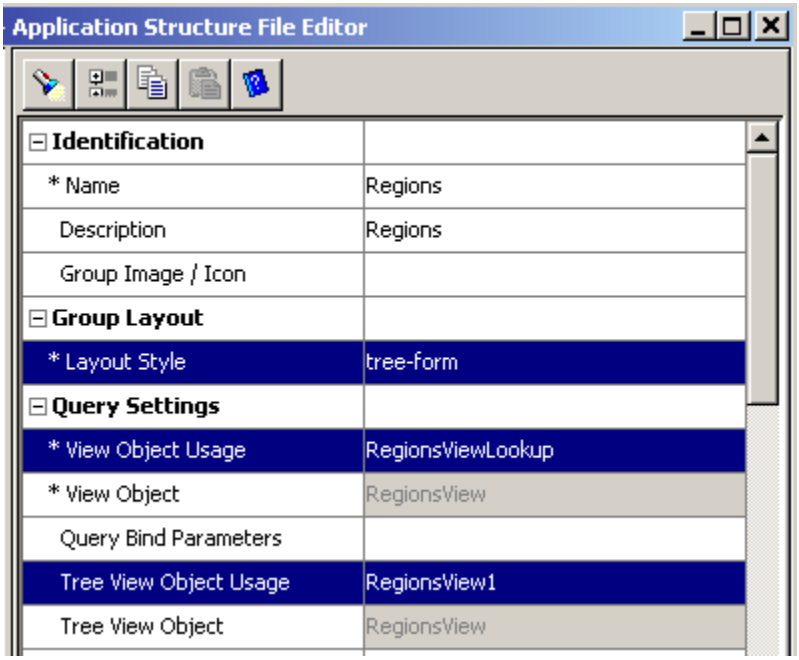
Before you can continue, check the DataModel of your Application Module. You should have this:



The same 5-level nesting structure should be present in your application structure file (generated by the New Application Structure File wizard):

1. For the Regions group and all its detail groups, change the **Layout Style** property to 'tree-form'.

2. For each of these groups, change the **Tree View Object Usage** to be the same as the generated value of the View Object Usage property, and change the **View Object Usage** property to the Lookup usage of that view (for example RegionsViewLookup, CountriesViewLookup, etc).



3. Run the JHeadstart Application Generator. You will get something like this:

You can use the tree control to drill down in the hierarchical structure.

You can edit records on each level. JHeadstart has added a maintenance page for each level in the tree. You can navigate to the maintenance page by clicking on the hyperlinks in the tree.

CHAPTER

3

JHeadstart Application Generator

This chapter describes in detail the generation features of JHeadstart. It contains the following sections:

- Architecture describes the overall picture of JHeadstart.
- Roadmap describes the development process when using JHeadstart.
- Using the JHeadstart Addins describes how to use the JHeadstart tools.
- Prepare Model for generation describes the recommended preparation steps in the ADF BC Model layer.
- Page Layout Generation describes the layout styles JHeadstart can generate.
- Query Behavior describes the search features of JHeadstart.
- Transactional Behavior describes how to influence insert, update and delete capabilities of a page.
- Generating User Interface Widgets describes the item types JHeadstart can generate.
- Customizing Page Layout Generation describes how you can adapt the JHeadstart generation process to your own needs.
- Internationalization describes how to handle translation topics.
- Security describes how you can generate authorization features such as restricting Group access and read-only behaviour into your application.
- What Was Generated for what purpose describes the output files of the generator.

How to use this chapter

Continue with reading the Architecture section and the Roadmap. The Roadmap has links to relevant sections in the document.

Architecture

This section describes the high level architecture of the JHeadstart Application Generator (JAG).

The JHeadstart Application Generator provides a simple, highly productive means for creating a transaction-based J2EE application using ADF.

The Application Structure File drives the JHeadstart Application Generator. This is an XML file that defines the overall structure of the application, including:

- The type of view layer that should be generated (UIX/JSP).
- The View Objects that should be displayed and modified.
- The layout styles that should be used to display and manipulate the View Objects.
- Relationships between the View Objects: master-detail or lookup.

JHeadstart is delivered with a JHeadstart Application Structure File editor, which is a user-friendly mechanism to edit the Application Structure without having to edit the XML file directly.

Input Output

In addition to the Application Structure File, the Application Generator uses the following inputs:

- A Domain Definition File that contains the definitions of static domains.
- ADF Entity Object XML files.
- ADF View Object XML files.
- JHeadstart Generator Templates.

The Application Generator parses this Application Structure File and generates an MVC application using the following technologies:

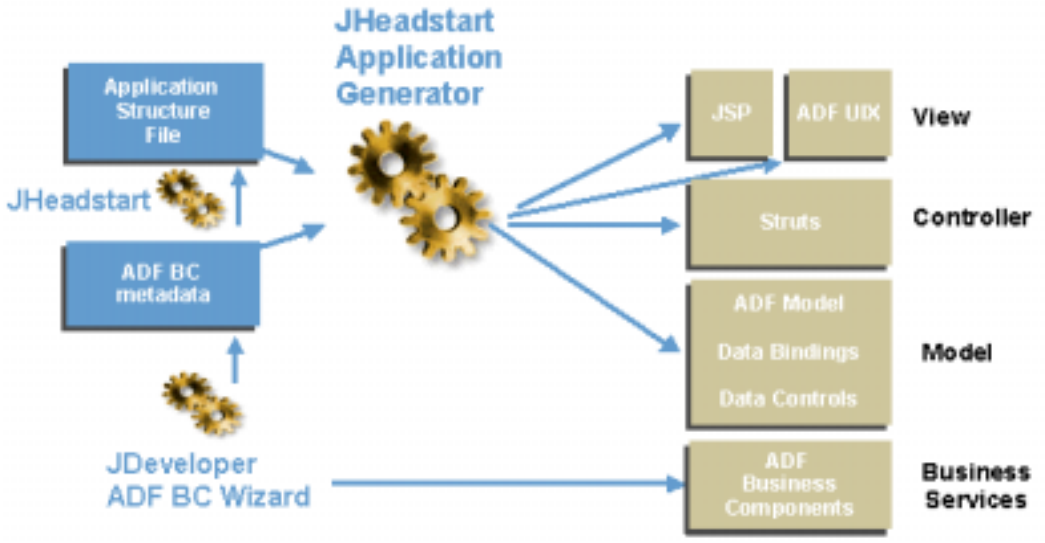
- Model: ADF Business Components and ADF Model.
- View: User Interface XML (ADF UIX) or Java Server Pages (JSP).
- Controller: Struts.

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

- Struts Config file for the Struts Controller.
- UIX/JSP files for each displayed page.
- Page UI Model for generated pages.
- Resource bundles for internationalization.

The output of the JAG, together with ADF Business Components forms the complete J2EE application.

Whenever it is required, you can switch on and switch off generation of individual file types.



JHeadstart Application Generator

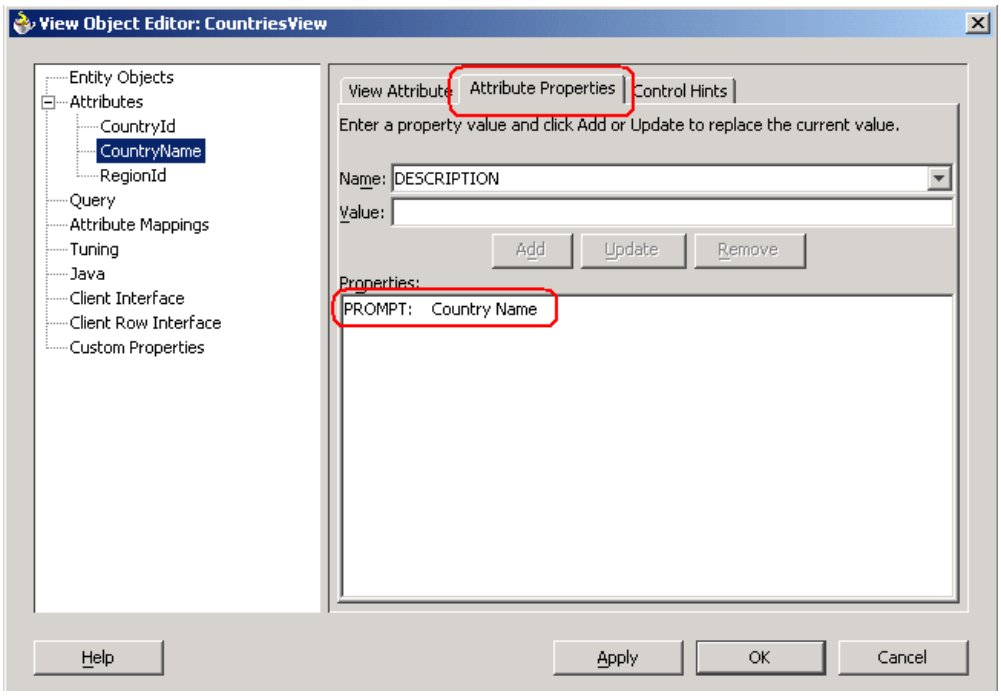
Use of BC custom properties

The JHeadstart Application Structure File contains a high-level description of the application you want JHeadstart to generate. You maintain this file with the Application Structure File Editor.

JHeadstart also stores information in the Custom Properties of ADF BC View Objects and Entity Objects. For example, the prompt of an attribute is stored in the Custom Property PROMPT.

There are two approaches possible for setting the custom properties:

1. Using the JHeadstart ADF BC Property Editor. The JHeadstart Property Editor is an add-on specifically created to enter these kinds of properties. In the descriptions below it is assumed that you use the JHeadstart ADF BC Property Editor. See Using the ADF Business Components Editor
2. Using the JDeveloper standard editors. Go to the View Object, right mouse click, and select Edit <View Object>. Select an attribute and select the Attribute Properties tab. There you can see the defined custom properties for the attribute.



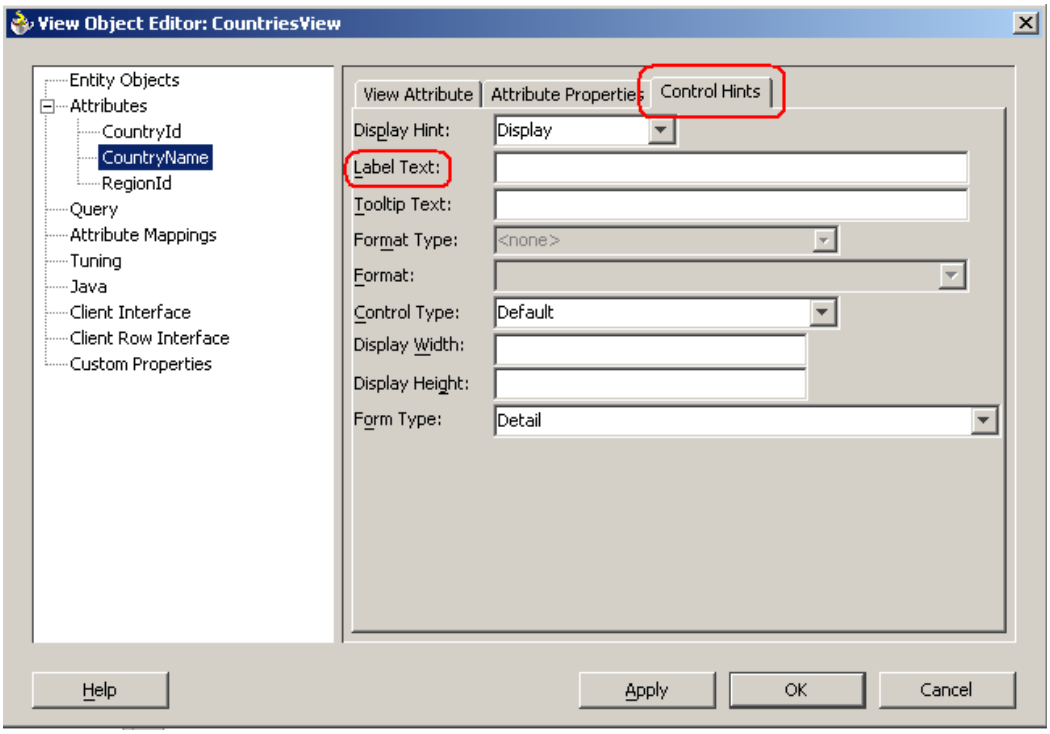
It is strongly recommended to use the JHeadstart ADF BC Property Editor because it is much easier to use. The JDeveloper standard editors are not so user-friendly for maintaining lots of JHeadstart Custom properties.

Suggestion: Look into a <ViewObject>.xml file directly. You will see the custom properties there. Example:

```
<ViewAttribute
   Name="CountryName"
   PrecisionRule="true"
   EntityAttrName="CountryName"
   EntityUsage="Countries"
   AliasName="COUNTRY_NAME" >
   <Properties>
      <Property Name ="PROMPT" Value ="Country Name" />
   </Properties>
</ViewAttribute>
```

BC Control Hints

You can also specify Control Hints for an attribute in a View Object. See the screenshot below.



Information entered here gets generated by BC in a resource bundle. Some of the Control Hints of BC overlap with custom properties used by JHeadstart. For example, 'Label Text' overlaps with the custom property 'Prompt'. When you did not define a custom property, the JHeadstart Application Generator will use the Control Hints.

Roadmap

When you want to develop an application using the JHeadstart Application Generator (JAG), you will have to follow the steps below:

1. **Set Up JDeveloper Workspace and ADF Business Components Package**
   Before you can run the JAG you must apply the steps of Chapter 2, *Getting Started*. If you migrate an application from Oracle Designer you will typically also run the JHeadstart Designer Generator (JDG) first. See Chapter 4, *JHeadstart Designer Generator*, for more information.
2. **Prepare your Business Components Model.** You should get your model right, before starting to generate with JHeadstart. See section Prepare Model for generation.
3. **Enable JHeadstart for the project and create an Application Structure File.**
   Again see Chapter 2, *Getting Started*.
4. **Determine the Service level properties of the Application Structure File**
   Consider which View-Type best matches your needs (see also Chapter 1, *Overview*). Specify the desired type in the Application Structure File.
   Check the other Service attributes to see if you need to change the default settings.
5. **Define your Groups in the Application Structure File**
   The Application Structure File contains a number of group definitions to determine the pages you need in the application, the layout of these pages, what kind of lookups you need and so on. For more details about the Application Structure File and its groups, see the section Using the Application Structure File Editor.
   If you use the JDG the Application Structure File will be generated for you based on the Module Component Definitions in Oracle Designer.
6. **Generate the page(s) using the JHeadstart Application Generator**
   When you have done the appropriate preparations, then you can start generating pages using the JAG. You do not have to have every detail ready before you use the generator. It is recommended that you start using the generator as soon as you've created your first group in the Application Structure so that you soon learn how the JAG works. See Running the JHeadstart Application Generator.
7. **Modify Entity and modify/add View Objects to improve page layout and behavior**
   The JAG uses the Entity and View Objects to determine aspects of the layout and behavior of the pages. You can create new View Objects with just exactly the attributes you need for the page and use those as the basis for your pages, and/or you may want to change the default View Objects, as you need them for the page.
   You can also add JHeadstart properties to the EO and/or VO attributes specifying various display properties, such as the prompt, display type (checkbox, choice etc). You can use the JHeadstart property editors to enter and modify the EO and VO properties, and the properties of their attributes. How to modify the objects in order to achieve the desired behavior is described in the various sections below.
   If you use the JDG there will have been created specific View Objects from the Module Components in Oracle Designer. These View Objects also have included the specific custom properties to – as much as possible – have the same kind of display settings as specified in Oracle Designer.

See Page Layout Generation and Generating User Interface Widgets for more details about layout generation

8. **Define Query and Transaction Behaviour.** See Query Behaviour on how you can generate the various search regions. See Transactional Behaviour on how to define Data Manipulation capabilities in generated pages.

9. **Change the Look and Feel of your Application**
   You can influence many aspects of the Look and Feel of your application while still doing 100% generation. See Internationalization for generating in different languages. For both JSP and UIX, JHeadstart has templates as a starting point for generation. So you can customize the Look and Feel by changing the templates and regenerate. See Customizing Page Layout Generation for details.

10. **Repeat step 4 through 9 until you are satisfied with the result**
    When you create an application using the JAG it will be natural to use an iterative approach to converge into the final version of your application. It is likely that you will have to make some modifications that cannot be generated using JAG, but it is recommended that you postpone this until you have generated your application as far as you can get with the JAG. This is to prevent having to repeat the post generation steps if you need to regenerate with the JAG. There are some switches that you can set to false to prevent the JAG from regenerating certain files, such as specific UIX pages etc.

11. **Perform any post-generation steps**
    To finalize your application, you will probably want to perform post-generation steps. As JHeadstart generates a 'normal' ADF application, you can take the generated application and make changes using all the development features of JDeveloper, like drag-and-drop, Visual Editor and so on.

Reference: See Oracle by Example Series: Oracle JDeveloper 10*g* on http://www.oracle.com/technology/obe/obe9051jdev/index.htm for tutorials on how to build ADF applications with JDeveloper.

Prepare Model for Generation

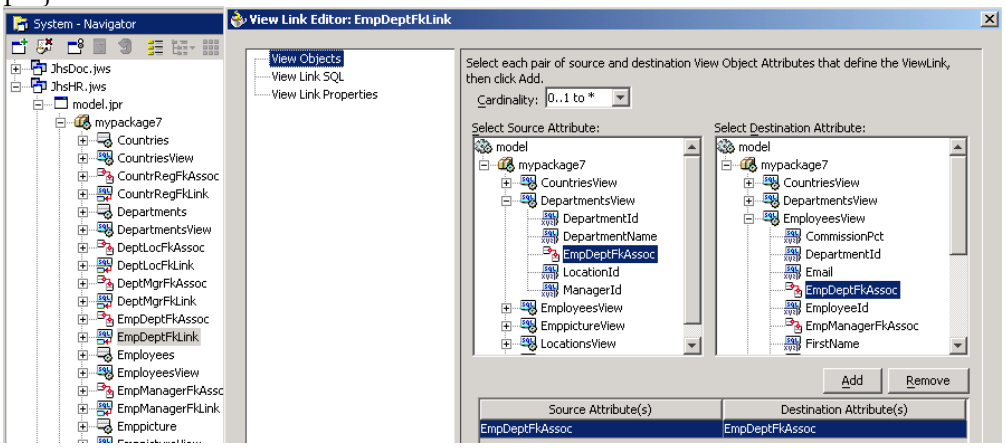
JHeadstart depends on information stored in the Business Components. It is therefore recommended to review the Business Components of your project, before creating the initial Application Structure File. By doing so, the New Application Structure File wizard will create a better structure file and your first generation will be of higher quality.

Setting up master-detail synchronization

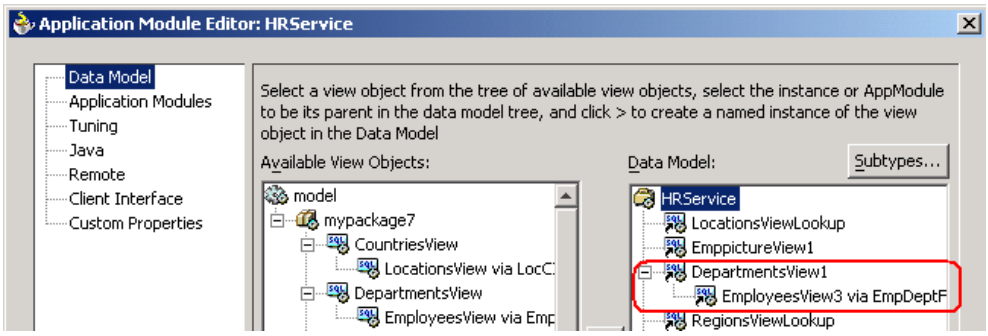
JHeadstart is capable of generating master-detail layouts. For example you want to show a departments with all the employees in that departments as detail.

When you want to generate master-detail layout, it is important to make some preparations in the ADF BC Model. Let's take the Departments with Employees as an example:

1. A View Link representing the master-detail relation must exist in your Model project:



2. The master-detail relation must exist in the Data Model of your Application Module

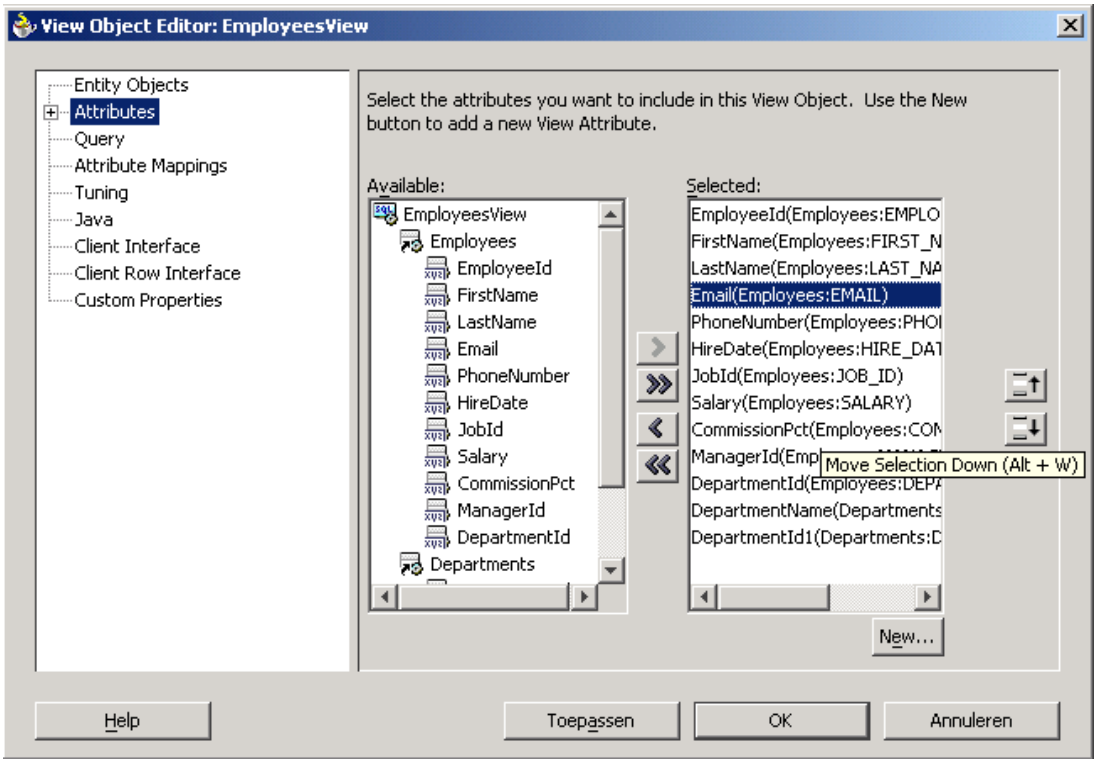


Attention: You can have more levels of nesting. For example Regions, consisting of Countries, consisting of Locations and so on. See section Creating Tree Layouts for an example of deeper nesting.

Determine the Display Sequence of Attributes within a Row

The display sequence of the items on the generated pages is determined by the attributes as they have been defined in the View Object you have specified for your group.

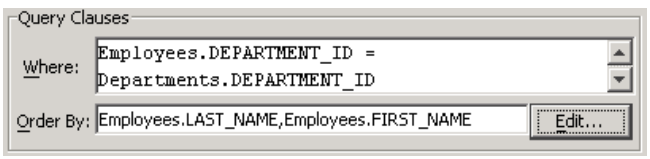
For each View Object, you must therefore define the order in which the attributes will be displayed on the page.



1. Select the View Object, right click and select Edit <view object>
2. Go to the Attributes node. The order in which the attributes are shown in the Selected list is the order in which they will be displayed on the page. (Attributes that are not displayed are discussed below.)
3. Use the up and down arrows on the right side of the screen to reposition attributes.

Determine the Order of Displayed Rows

In most situations you want to order the queried records. To accomplish this you must add an Order By clause to each View Object.



Attention: In general, there is NO DEFAULT sort order you can rely on.

1. Select the View Object, right mouse click, select Edit <ViewObject> to open its Properties dialog.
2. Go to the Query node and enter the Order By clause. You can press the Edit button to select available attributes. Often, the view is ordered by the Descriptor attribute. It may also be ordered by a lookup attribute.
3. Be sure to use the 'Test' button to verify the query.

It is also possible to let the user order the records as desired in a page with table format. See the section Allowing the user to sort data in a table page for more detail on how to do this.

Create Calculated or Transient Attributes

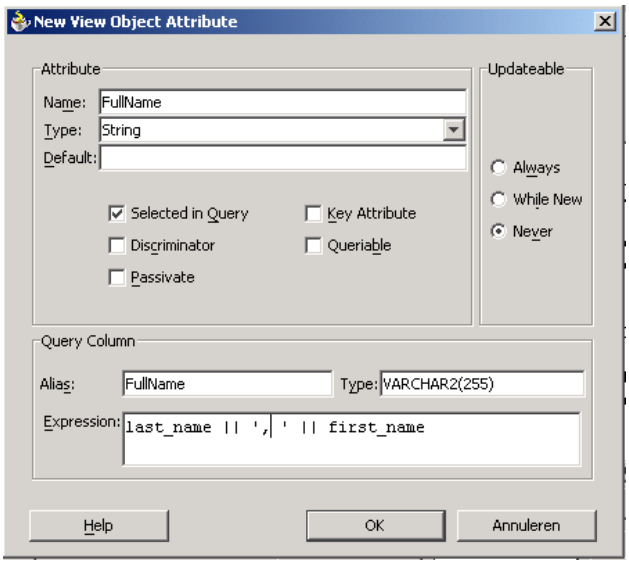
Sometimes you want to show an attribute that does not exist in the correct form in the database. For example: you want a read only attribute FULL_NAME based on the FIRST_NAME and LAST_NAME attributes. In such cases, you need to add a calculated or transient attribute.

Note the important difference between a calculated and a transient attribute:

- A calculated attribute is present in the SQL query: the calculation is done by SQL at retrieval time. So a calculated attribute is only recalculated when the query is reexecuted. Imagine a calculated attribute FullName that is a concatenation of FirstName and LastName. When the FirstName is changed in the application, the data needs to be requeried to refresh FullName. Only use a calculated attribute for readonly fields.
- A transient attribute is not present in de SQL query. You have to calculate the value in a get method in the View Object. Every time the transient attribute value is needed, the get method is called and the transient value is recalculated. So you have no synchronization issues when using a transient attribute. The only drawback of a transient attribute is that you have to code a get method in the ViewRowImpl class.

Steps to create a calculated attribute

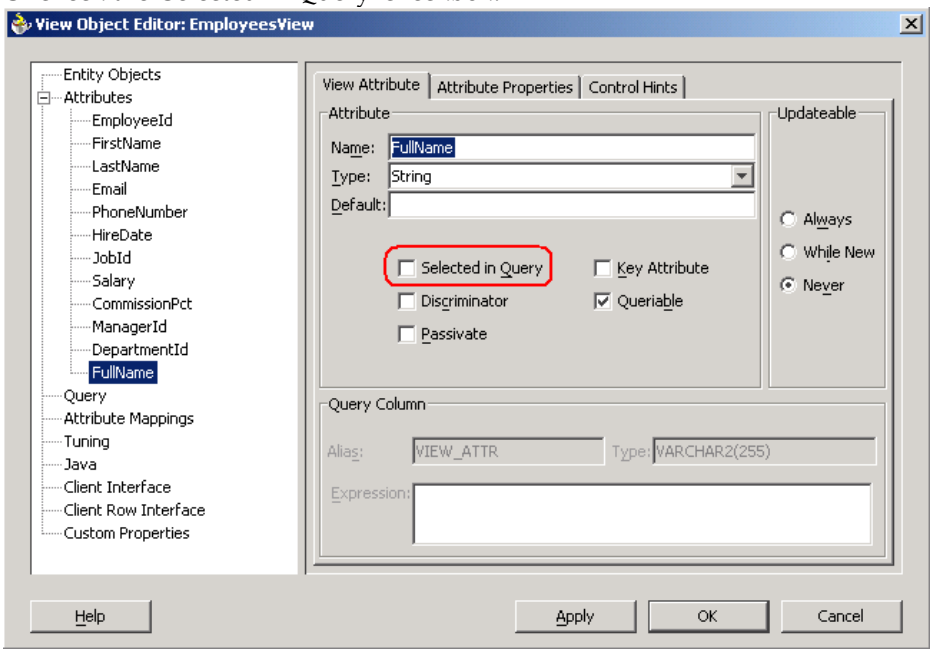
1. Select the View Object, right mouse click, select Edit <ViewObject> to open its Properties dialog.
2. Go to the Attributes node and click on 'New'.
3. Enter an appropriate name for the attribute.
4. Check 'Selected in Query'.
5. Give the attribute an alias.
6. In the Expression, key in the SQL expression to create the concatenated string. Note that if you have included lookup attributes in your view definition, then you must include the alias you have given to the selected entity objects in the SQL Expression:
   For example: EMPL.LAST_NAME || ', ' ||EMP.FIRST_NAME



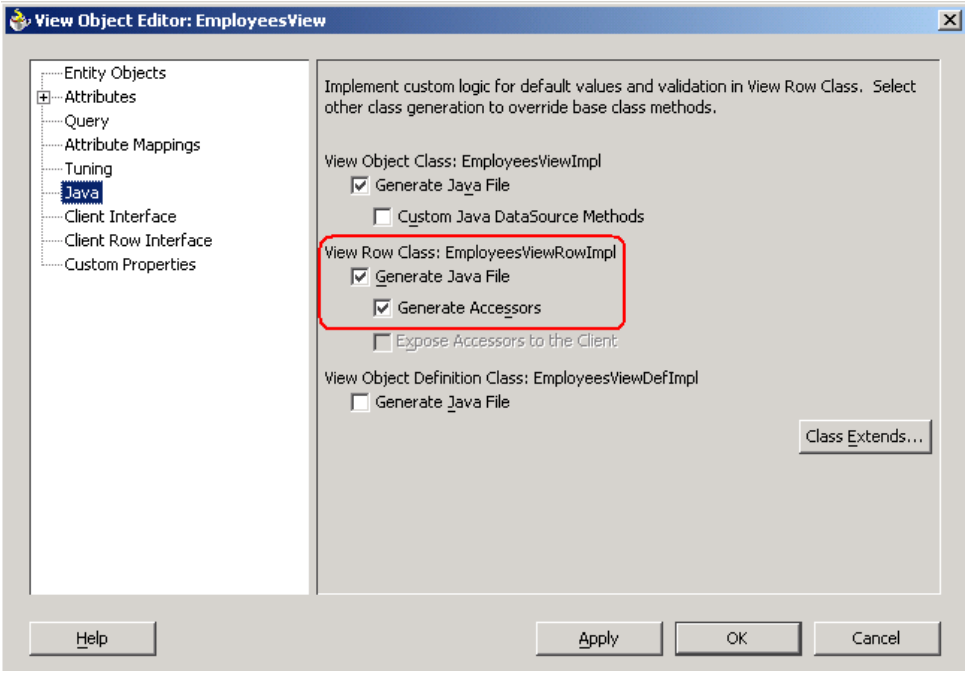
7. Position the new attribute in the desired place on the attribute list to ensure it is displayed at the correct location on the page.
8. Save the changes and close the View Object dialog.

Steps to create a transient attribute

1. Select the View Object, right mouse click, select Edit <ViewObject> to open its Properties dialog.
2. Go to the Attributes node and click on 'New'.
3. Give the attribute an appropriate name.
4. Uncheck the 'Selected in Query' checkbox.



5. Go to the Java page.
6. Ensure that Generate Java File is checked for the View Row Class, and that Generate Accessors is checked.



7. Press OK
8. Open the generated ViewRowImpl.java file (open the View Object node to see the file, or check the Structure Pane) and go to the get<newAttributeName> method.
9. Code the 'get' method for this attribute in the view java class. For example:
```
public String getFullName()
{
 return getLastName() + "," + getFirstName();
}
```
10. Position the new attribute in the desired place on the attribute list to ensure it is displayed at the correct location on the page.
11. Save the changes and close the View Object dialog.

Reference: It is recommended to test the ViewObject with the Business Components Browser. See section Test the model

Generated Primary Key Values

In many cases, artificial primary keys are used (also known as surrogate primary key). Typically, these primary key columns are called ID. Because they are artificial, they are meaningless to the user. The system generates the values and uses them internally, but they should be hidden for the user.

Before starting to generate applications with JHeadstart, examine your Model for artificial primary keys. Make sure they are correctly populated. Test this with the Business Components Application Module Tester. See section Test the model

An artificial primary key can be populated in two ways:

- In the Business components: The create method of the entity object is used for that.
- In the database: A database trigger is in added to the table that gets the next value from a database sequence and populates the primary key.

When you plan to create screens that insert a master row and one or more detail rows in one transaction, you will need to populate the primary key in business components, otherwise ADF BC will not be able to set the foreign key of the details rows correctly. Note that in this scenario, you can still use database triggers as well, if you have other non-ADF BC applications working on the same tables.

Surrogate primary key populated in the Business Components Model layer

In the Entity object implementation, a create method is added that takes the value out of a database sequence and sets the primary key.

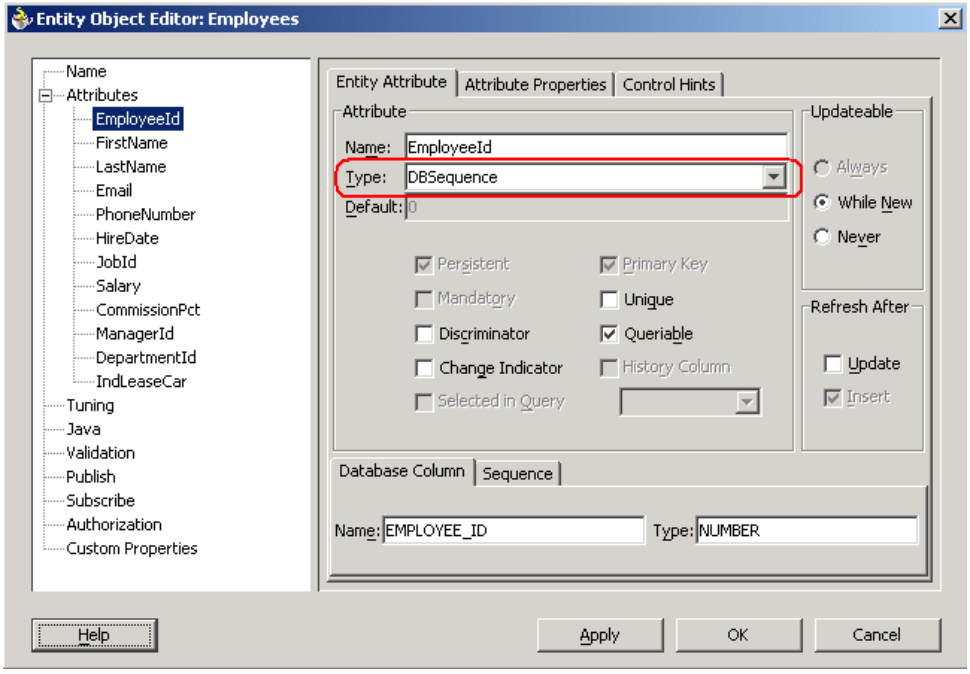
This is described in detail in the JDeveloper Help. Check topic 'Populating an Attribute from a Database Sequence'.

Suggestion: If all primary key attributes have the same name, for example Id, retrieving sequence values from the database in the create method is something you could implement in your BC base classes. By doing so, you do not need to implement a create method for each entity object. In the EO base class you can retrieve from one sequence that is used for all Entity Objects. Or you can implement a more sophisticated mechanism that derives the sequencename from the Entity Object name.

Surrogate Primary Key populated in the database

The database generates the primary key value, so no Java code is needed to populate the primary key. However, your Business Components Model needs to know that values get refreshed in the database after the insert.

1. Set the type of the attribute to 'DBSequence'. ADF BC will now derive an artificial key until the row is posted to the database.



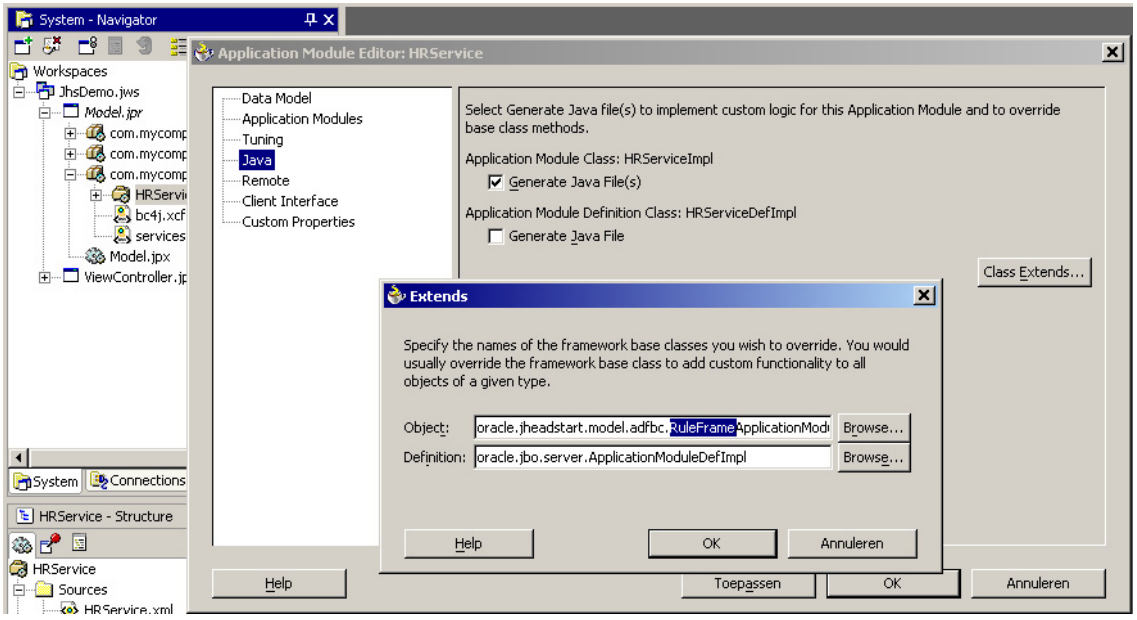
Using CDM Ruleframe

If you don't use CDM RuleFrame, skip this section.

If you use CDM Ruleframe for your business logic layer implementation, you will need a special extension class for your ADF BC Application Modules. This class won't be available in your Model project until you have run the JHeadstart Application Generator for the first time, because JAG will add the JHeadstart Runtime library to the Model project.

To enable CDM RuleFrame for an ADF BC Application Module:

1. Locate the cursor on your Application Module, right-mouse click, and select Edit.
2. Navigate to the Java node, and press the Extends button. Change JhsApplicationModuleImpl into RuleFrameApplicationModuleImpl, or alternatively, press the Browse button, navigate to oracle.jheadstart.model.adfbc, select RuleFrameApplicationModuleImpl, and press OK.
3. Press OK again, and finally OK to close the Application Module wizard. Choose 'Save all files' to save the changes.



Test the model

Before starting to generate with JHeadstart, you should be absolute sure you have your Model right. So make sure you can query, insert, update and delete data with your View Objects.

Use the shipped Business Components Tester for validating your model. Right-click your Application Module and choose 'Test…' . Check the Database Connection name and click the Connect button. Now the Oracle Business Component Browser opens.

On the left hand side you will see the Data Model of the Application Module. Double click one of the View Object Usages to open a browser for it. On the right hand side you can now browse through the rows, make changes to them, and, using the toolbar, even create and delete rows.

See the JDeveloper help for further instructions. Topic is 'Testing with the ADF Business Components Browser'.

Suggestion: This Tester application is a very convenient way of checking whether you have correctly specified your Business Components, without having to create a full-blown application on top of it first. Also, in multi-layered applications such as these, the exact source of a problem is not always easy to determine. The Tester application is also very useful in determining whether a problem is located 'above' or 'below' the ADF Bindings. Finally, it is a quick and easy way to test virtually any business rule implementation that was implemented in the Business Components.

Using the JHeadstart Addins

This chapter describes the terms and components used when working with the JAG. You will also learn how to use the JHeadstart Application Structure File Editor and the other JHeadstart addins.

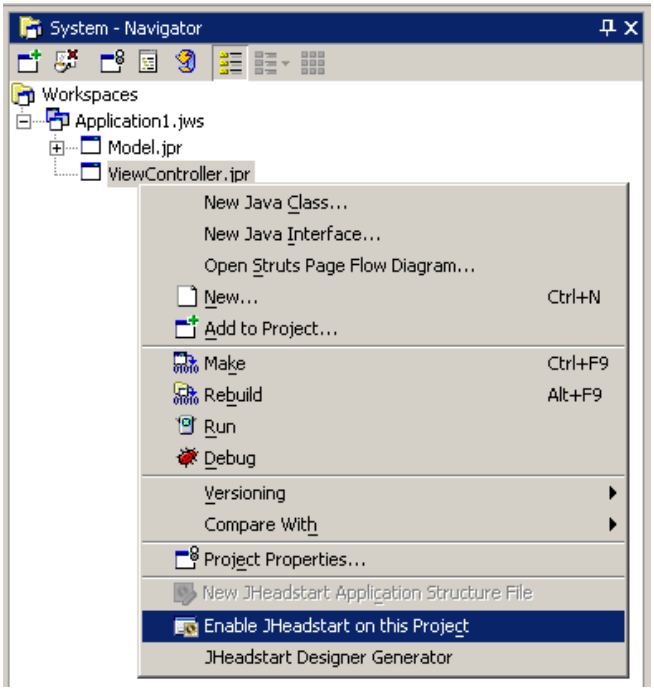
Enabling JHeadstart Wizard

JDeveloper offers a host of technologies that you might or might not use in a project. The use of some of these technologies might require the presence of some files or settings in your project. To facilitate the development process, JDeveloper will usually create these files and/or settings for you the first time you use such a technology in your project, often without notice. For instance, the first time you create a UIX page in a project, JDeveloper will automatically add a number of settings to the web.xml file, add a uix-config.xml file to your project, and also create an entire directory tree named 'cabo' under the HTML root directory of your project, containing runtime sources such as images and javascript libraries.

In a similar fashion, the use of JHeadstart also requires such files and settings in your project. We have chosen to make the use of JHeadstart on a project adeliberate choice. Before allowing the use of any JHeadstart Addins on a project, you must first 'enable' JHeadstart on it. Typically, this only needs to occur on the 'ViewController' project: the project that will hold the Struts Page Flow Diagram and the JSP or UIX pages. This action will also trigger the creation of those files and settings needed for a JHeadstart application.

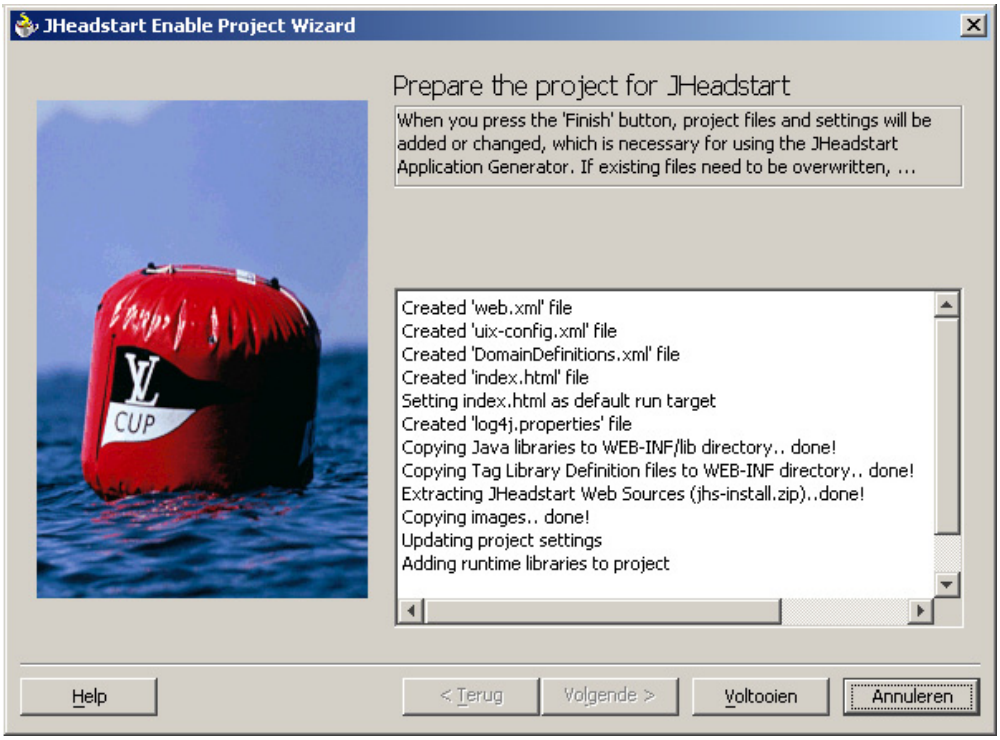
Enabling JHeadstart on a new project

Enabling JHeadstart on a project is a simple operation, that you can perform by right-clicking on the project, and selecting the option 'Enable JHeadstart on this Project'.



Attention: As you can see, the option 'New JHeadstart Application Structure File', which would invoke the JHeadstart wizard described in section 'Create New Application Structure File Wizard' is disabled at this time, because JHeadstart is not yet enabled on this project.

The Enable JHeadstart Wizard that is invoked by this menu option does not ask for any input. All you need to do is click 'Next' and 'Finish'. It will then create and add a number of files to your project, and make some required project settings. It will report what it has done in the following dialogue:



Enabling JHeadstart on an existing project

The above screenshot is the result of invoking the Enable JHeadstart Wizard on new project. But it is safe to use this wizard on a project that already contains many files, possibly even a fully functional ADF web application. That is because, unlike JDeveloper, this wizard will never overwrite any files or settings without either backing them up or asking for your feedback on how to proceed. To be more specific, here are the possible responses of the wizard when trying to create a file that already exists:

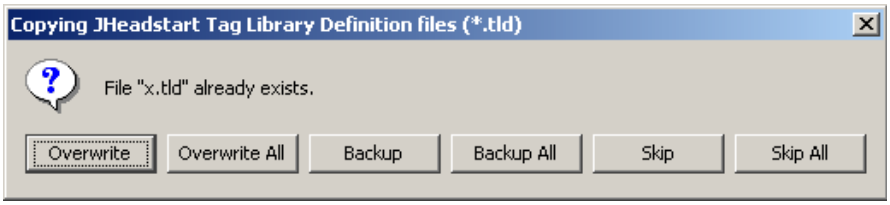
1. Backup the file.

   This is done for files that are absolutely required, for JHeadstart to function correctly, such as 'web.xml' and 'uix-config.xml'. **If you made manual changes to these files, you will need to merge them from the backup to the new version created by JHeadstart!**

2. Ignore the file and keep the existing version.

   This is done for less vital files such as index.html and log4j.properties

3. Prompt for your resolution.

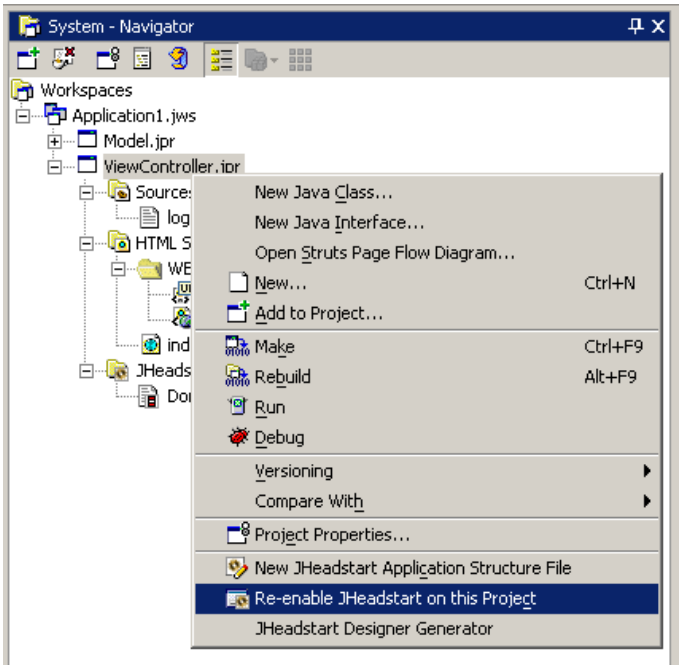


This is done for all other files, such as Tag Libraries and JHeadstart-specific files. It is unlikely that you made manual changes to these files, so normally you would choose 'Overwrite All', but you can make your choice to overwrite, backup or ignore on a per-file basis if you want.

Re-enabling JHeadstart on a project

Because of the safe nature of the wizard, we have allowed the option to re-run the wizard on a project that you have already used it on. You can do this, for instance, if you receive a newer version or patch of JHeadstart and want to make sure you are using the latest runtime files, or if you have made changes to the files that you want to undo by reverting back to the original version.

To rerun the wizard again, right-click on the project and choose 'Re-enable JHeadstart on this Project'.

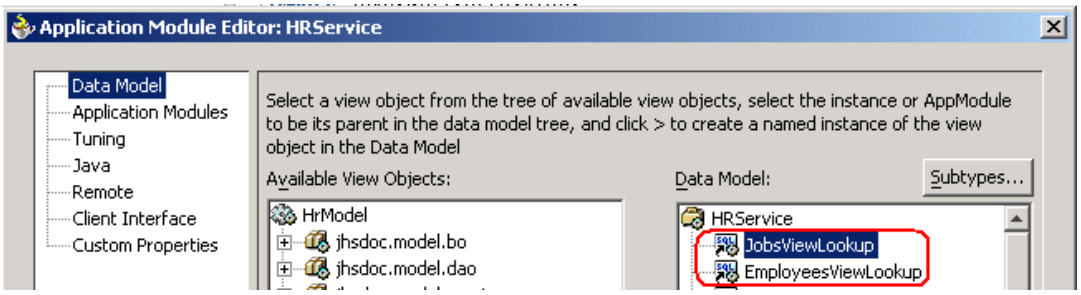


Attention: As you can see, because JHeadstart was already enabled on this project, you can choose the menu option 'New JHeadstart Application Structure File', and the menu option to launch the Enable JHeadstart Wizard was renamed to 'Re-enable JHeadstart on this Project'.

Create New Application Structure File Wizard

After enabling JHeadstart, you will add a new JHeadstart Application Structure file. At this point, JHeadstart will make changes to your ADF Business Components Model.

For each View Link, JHeadstart generates a Lookup by default. JHeadstart adds new instances of the ViewObjects to the Application Module with name *ViewLookup. You can inspect this behaviour by editing your Application Module.



The reason is that a lookup needs to maintain its own set of rows. For example, when you have a page that maintains Employees, and in another page there is a list of values for selecting an employee, there need to be two instances of the same ViewObject. One instance holds the rows for the maintenance page, and the other holds the rows for the list of values.

Using the Application Structure File Editor

Application Structure File

The Application Structure File defines the structure of your application. It identifies which pages you need, how you want these pages related, their layout styles, what information sources they are based on, and so on.
Each group in the application will be generated as a tabbed page in your web application.

One Application Structure File can contain only one Service. If you need multiple Services, you must create multiple Application Structure Files.

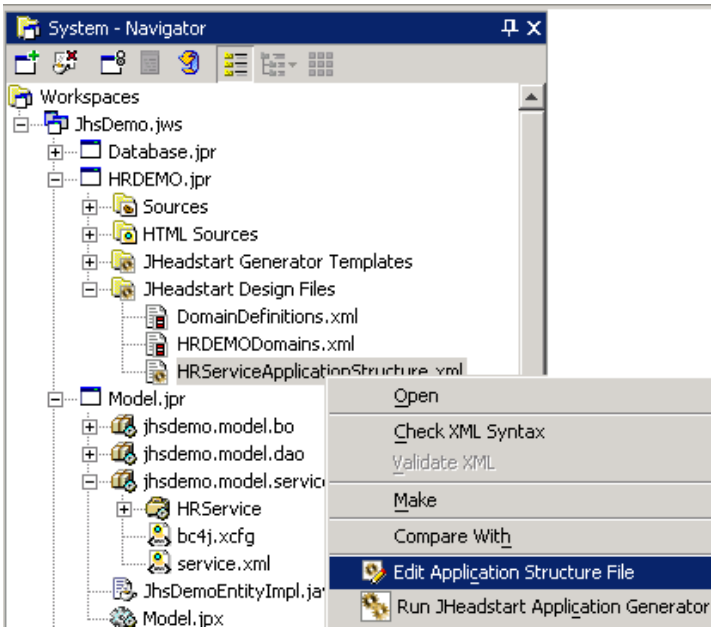
Maintaining the Application Structure File

The JHeadstart Application Structure File Editor helps you to maintain the Application Structure File without having to write and edit the XML yourself. You simply define or modify the properties as you need, and the XML file will be modified accordingly.

Reference: How to create an Application Structure File and how to define a service is described in Chapter 2, *Getting Started*. This section only discusses how to create new groups, and how to modify and remove existing groups.

Starting the Application Structure File editor

To be able to start the editor you must have created an initial Application Structure File. Place the cursor on this file, and press the right-hand mouse click:

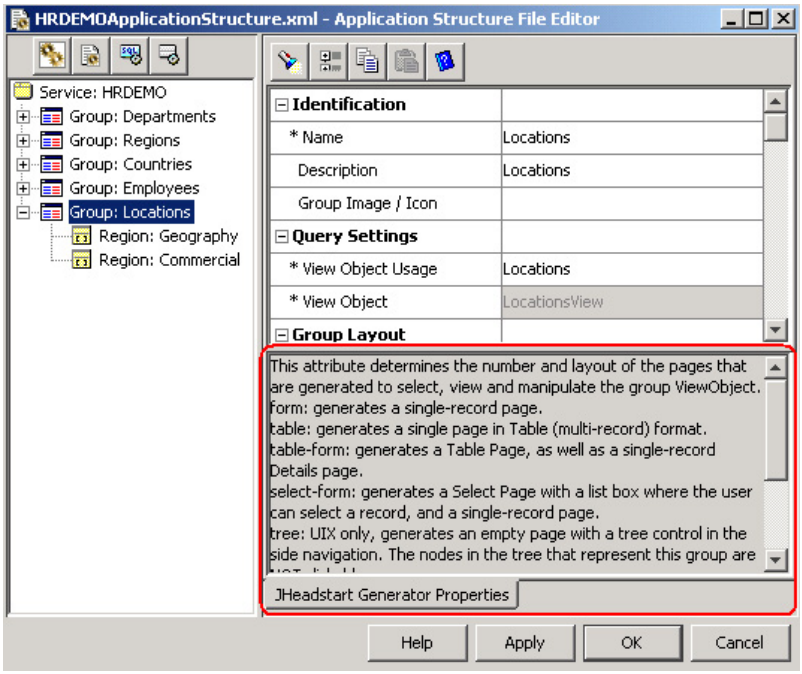


Select 'Edit Application Structure File' to open the editor.

Using the help in the Application Structure File editor

The help in the Application Structure File editor explains all the properties that you can set for each service, group, detail group, lookup and region. This is a very useful aid to help you determine how and when to set each property.

When you open the Application Structure File editor, you will see the properties on the right hand side of the editor. Below the properties, you see a small area (enlarged in the screen shot below) with the help text. If you click on a property, the help text appears in the window for that property:

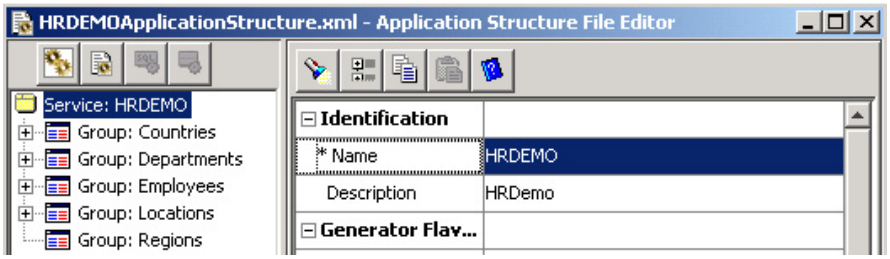


This area may seem unnecessary small. You can increase or decrease the size of this area as you desire, just by placing the mouse cursor on the line above the text and move the line up or down.

Service

A service must be seen as a major subset of the application. It includes a set of logically related functionality on which a user performs tasks that are logically linked together. The whole functionality will be displayed to the user on a number of tabbed pages accessed via a common tab bar.

A part of a service definition seen through the Application Structure File Editor



And the generated tab bar:

Attention: When partitioning the application into services, there are some restrictions to take into account:

1. A service can only be related to one ADF BC Application Module. However you can use one Application Module for multiple services.

2. Currently, JDeveloper can only have one Struts page flow diagram per project. However, it is possible to generate from multiple services to one Struts Config file, assuming the Group names are unique over all services. You might consider splitting up your project in multiple projects with each project having its own Struts Config file.

Reference: There is How-To on Oracle Technet for setting up multiple Struts Configurations. See 'How To Use Multiple Struts Configurations With JDeveloper 10*g*' on http://www.oracle.com/technology/products/jdev/howtos/10g/StrutsMultiConfigs/struts_multiconfig_howto.html

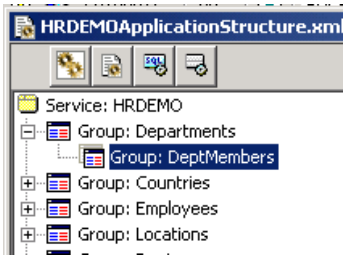
The JHeadstart Demo application included with JHeadstart also uses multiple struts-config files. Take a look at the web.xml and project settings in the jhsdemo workspace.

Group

A service is made up of one or more groups. A group allows users to query and modify a single ADF view object (VO). Depending on the layout options you choose, the group may be displayed on a single page or on a number of related pages.

Groups may be nested to support parent-child relationships between their respective View Objects.

Comparing to a form module defined through Oracle Designer, you would typically create one group for each first level module component. For detail module components in a master-detail relationship, you would use nested groups.

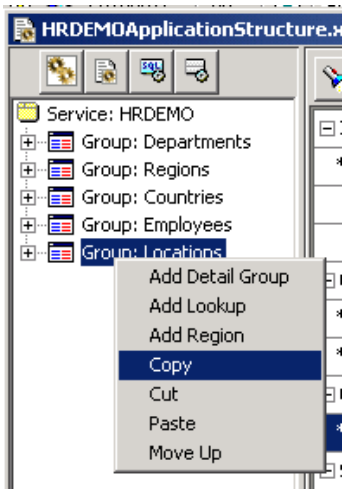


Creating a new Group

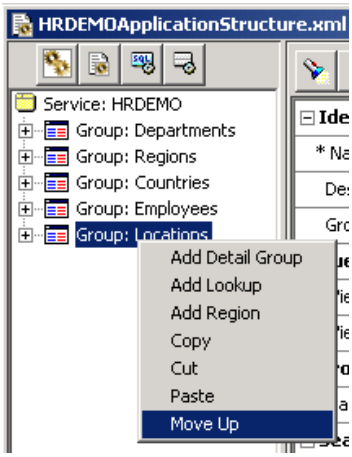
To create a new Base group locate your cursor at the service, and press the right-hand mouse button. Select Add Base Group, and you will get a new group at the bottom of your groups named newGroup. Some of the properties will be set with a default value. You can now enter all the properties as required for the group.

If the group you want to create should be identical, or almost identical to another group, then you can copy a whole group and create a new one that is identical to the one you copied. Thereafter, you can make any changes to the new group as required.

Simply select the group you want to copy, press the right-hand mouse, and select Copy:



If you want the group to be a main group, then locate your cursor at the Service definition, press the right-hand mouse, and select Paste. A new group is created at the bottom of all group definitions. If you want to change the order of the groups, you can select Move Up/Move Down in the right-hand mouse menu:

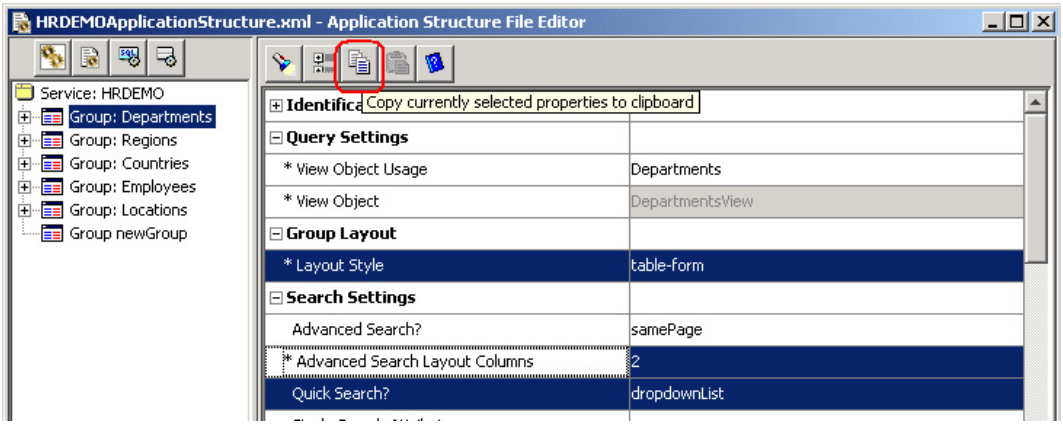


Attention: The order of the tabs of the generated application is determined by the order of the groups.

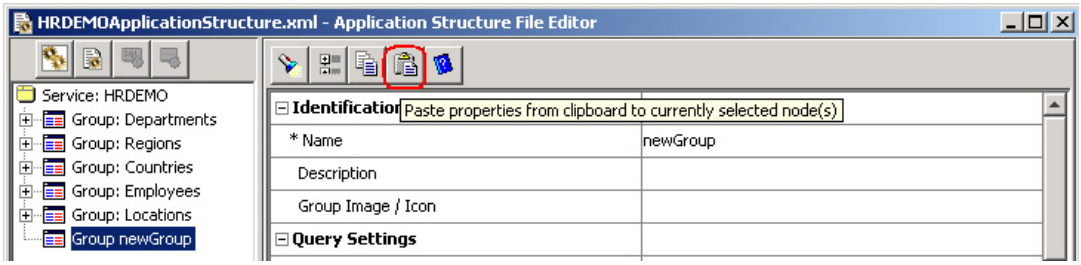
If you want the new group to become a detail of another group, simply place the cursor on the group you want the new group to become a detail of, and select Paste from the right-hand mouse menu.

Using the clipboard to copy and paste multiple properties

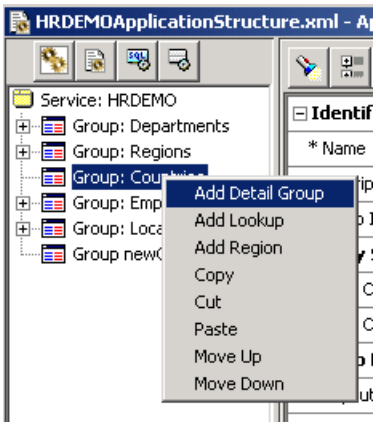
If the group should only be similar to another group, and you only want to copy a few properties, then you can also copy the properties you want from one group and paste them into the new group. Simply select the group you want to copy from, select all the properties you want to copy and press the button 'Copy currently selected properties to clipboard':



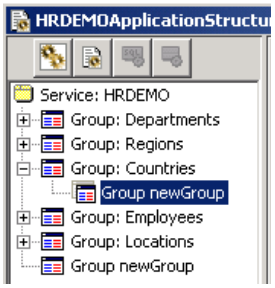
Then navigate to the group you want to copy to, and press the button ' Paste properties from clipboard to currently selected node(s)':



You can also create detail groups for base groups. Select the group you want to be the base group of the detail group, and press the right-hand mouse:



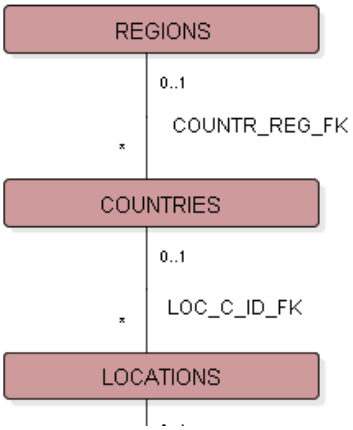
Select Add Detail Group. You will get a detail group named newGroup, and it is clearly visible in the left hand pane that the detail group is a detail to the selected base group:



Again, you can copy properties from other groups if many of the properties are identical.

Nested Groups

Groups can be nested to create master-detail (or parent-child) relationships.



In the Application Structure File editor, this looks as follows:

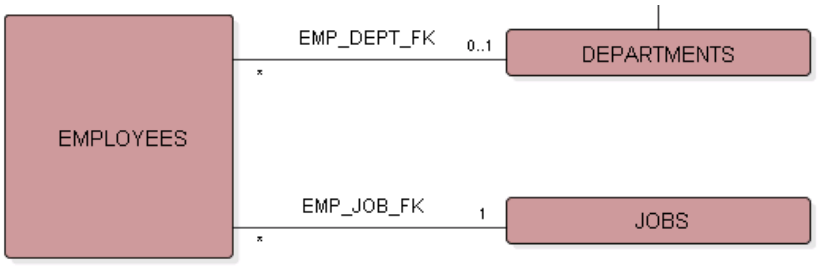
Depending on the layout options you choose, you can display a master group and its detail on different pages or on a single page.

Attention: JHeadstart has no restrictions on the maximum level of nesting. However, you can't have more than two levels of groups generated on one page.

Lookup

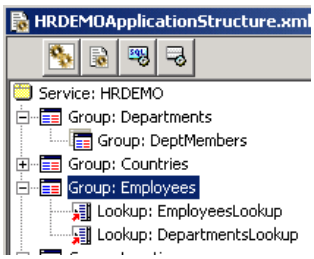
A lookup is a relation to another view object from the base view object that has been defined for the group. This is required if you need a list of values or a dropdown list.

Example `The EMPLOYEES table has two foreign keys.`



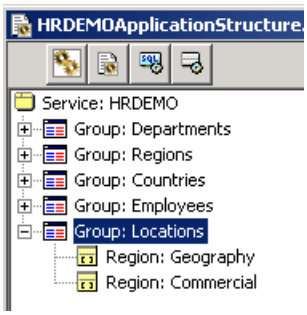
When adding an EMPLOYEE, you need to choose the department and the job. So you need two lookups, one for DEPARTMENTS and one for JOBS.

In the Application Structure File editor, this looks as follows:



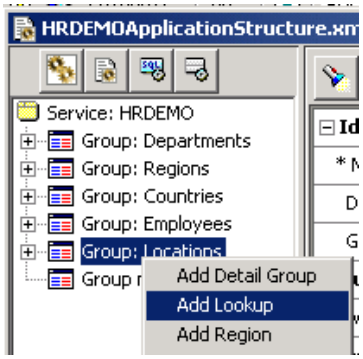
Regions

A Region is a subset of a group and allows you to group items into different sections (regions) on a page. You can define as many regions as you want for a group.



Creating a new Lookup or Region

You can create new Lookups or Regions using the JHeadstart Application Structure Editor. Select the group you want to be the base group of the detail group, and press the right-hand mouse:



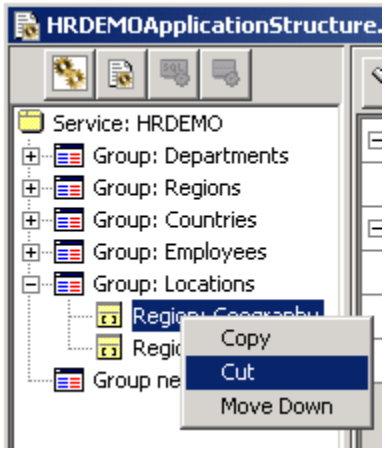
Select Add Lookup or Add Region. Modify the default properties.

You can create new lookups or regions by copying other lookups or regions as described for groups. You can copy selected properties from other lookups or regions if many of the properties are identical.

See section Generation of Lookups for changes you have to make in the ADF BC Objects when you want to use a Lookup.

Deleting a Group, Lookup or Region

You can also quickly delete a Group, a Lookup or a Region using the editor. Simply locate the cursor on the Group, Lookup or Region you want to delete, and press the right-hand mouse:



Press the Cut option, and the activated Group, Lookup or Region is removed.

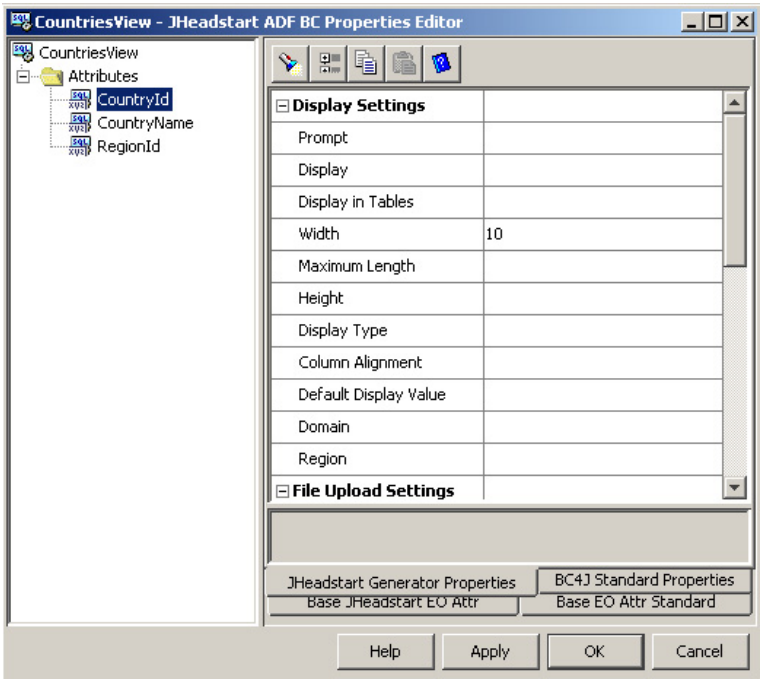
Using the ADF Business Components Editor

The JHeadstart ADF BC Property Editor allows you to specify the custom properties for an Entity Object or a View Object. This helps to quickly add and modify these properties, and helps to prevent simple typing errors. All the custom properties below can be entered through the JHeadstart ADF BC Property Editor.

There are two ways to start the JHeadstart ADF BC Property Editor:

1. Activate the Entity or View Object you want to add a custom property for in the JDeveloper System or Applications Navigator. Right-click and select JHeadstart ADF BC Property Editor.
2. When in the JHeadstart Application Structure File Editor, press the either the button with the View Object icon or the button with the Entity Object icon to launch the corresponding JHeadstart ADF BC Property Editor for the selected Group or Lookup. The buttons are disabled when the editor cannot determine the corresponding View- or Entity Object for the selected context.

When the editor opens, you can select the attribute you want to include the property for, and you will see all the properties at the right hand side of the editor:



If you view the bottom part of the right hand side of the editor, then you will see four tabs. This means that you can also view the Business Components Standard Properties, and update the ones that are updateable. You can also view the attributes as they have been defined on the underlying Entity Object if you are viewing a View Object.

Using the help in the JHeadstart ADF BC Properties editor

The help in the JHeadstart ADF BCJ Properties editor explains all the properties that you can set for each attribute. This is a very useful aid to help you determine how and when to set each property.

When you open the JHeadstart ADF BC Properties editor, you will see the properties on the right hand side of the editor. Below the properties, you see a small area with the help text. If you click on a property, the help text appears in the window for that property:

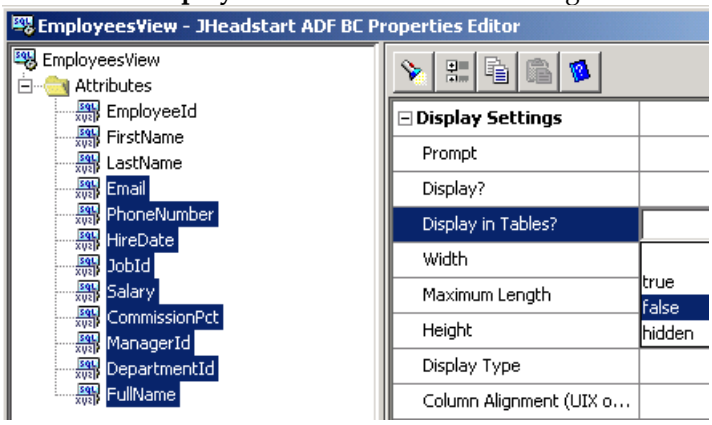
You can increase or decrease the size of this area, just by placing the mouse cursor on the line above the text and move the line up or down.

Using the clipboard to copy and paste multiple properties

Similar to the Application Structure File Editor, you can select one or more properties and copy them to the clipboard. This is useful when making the same changes to multiple attributes.

Using multi-select for changing multiple attributes

In many cases you need to change a property for multiple attributes. For example you want to set **Display in Tables?** to false for a range of attributes.

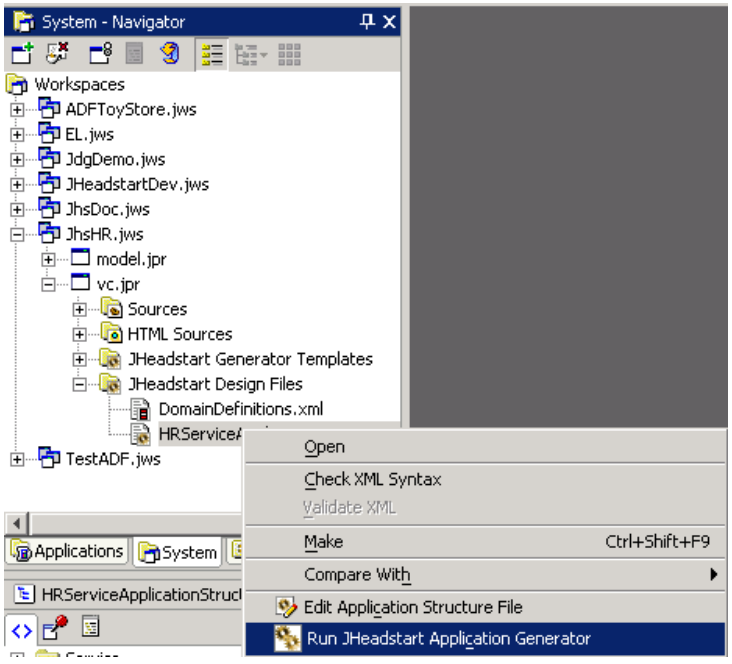


Running the JHeadstart Application Generator

When you have got the group definitions right, you can generate the application.

Perform the following steps to run the JHeadstart Application Generator:

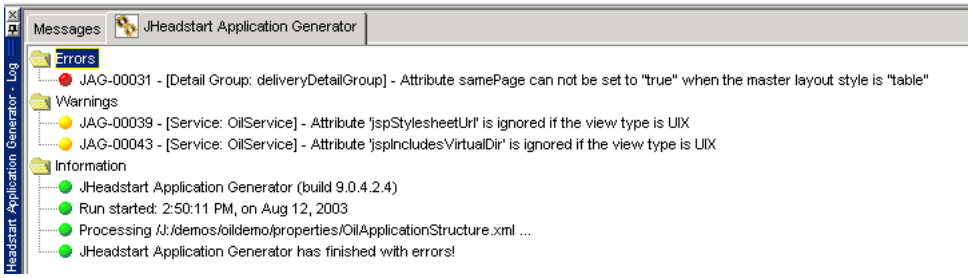
1. Navigate to your Application Structure File under the JHeadstart Design Files category of your project.
2. Right-mouse click, and select Run JHeadstart Application Generator



When the generator is running you will get messages in the JHeadstart Application Generator log window indicating what is generated.

You will see logging of what has been generated (the **Information** messages).
If the Generator ignores some settings, you will see **Warnings**.
If a problem is detected that does not prevent the Generator from doing it's job

but what could be a problem when you run the application, you will see **Runtime Warnings** displayed.
Finally, if something goes wrong during the generation process, you will see **Errors**.



3. When the JHeadstart Application Generator has completed successfully, you can run and test your application.

Reference: See 'Running the Application' in Chapter 2, *Getting Started*, for instructions on the different ways to run the generated application.

Page Layout Generation

This section describes the various layout styles JHeadstart can generate.

The following layout styles are described:

1. Creating Form Pages
2. Creating Select-Form Pages
3. Creating Table Pages
4. Creating Table-Form Pages
5. Creating Tree Layouts
6. Creating Shuttle Layouts

Creating Form Pages

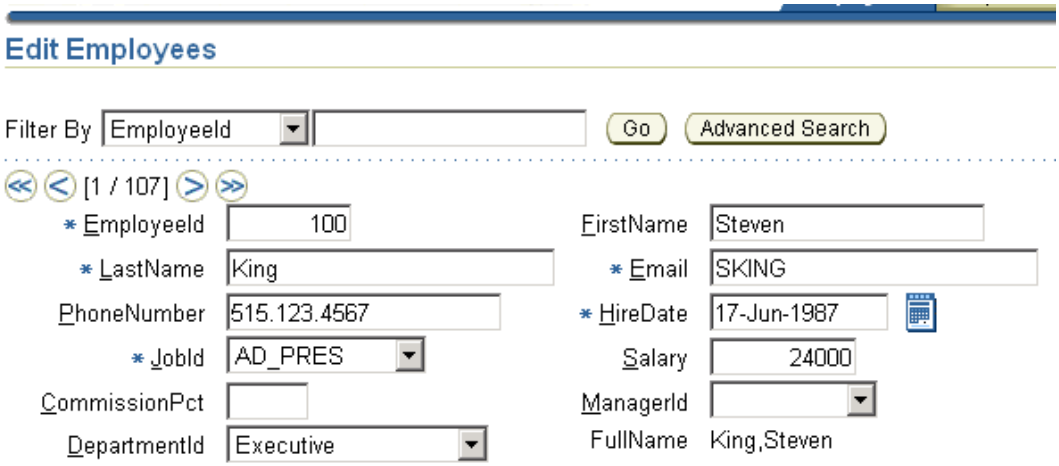
With a form page you can manipulate one row at a time. You typically use a form page when the records have many attributes and you want to show all of them.

Add the following attributes to your group definition n the Application Structure File to generate a Form Page:

1. Set the **Layout Style** attribute to 'form'.
2. Determine the amount of horizontal space the form layout can consume on the page as a percentage (**Form Width**). If you set this to 100%, the items will spread out over the whole page. However, the items will not be aligned with each other, but will be spread over the page to take the full space available.

   If you want to force the items to be left aligned, set the value to an arbitrarily small number. At runtime, the screen painter will see that the value is too small and automatically increase the width just enough to display the items left aligned. The default is 10 which will left align the items.
3. Determine the number of **columns** used to layout the fields in a form layout style. The default is 1, which will leave a page with all items placed below each other in one column.

Example of a form page:

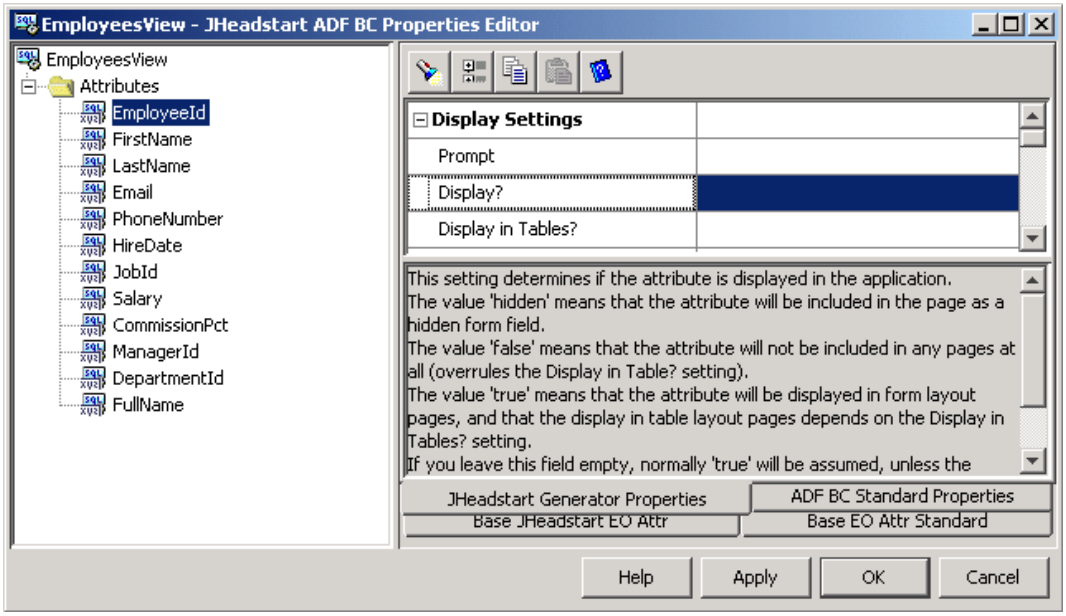


Notice that the attributes are laid out in two columns as specified in the **columns** property.

Attention: The display sequence of the attributes on the generated page is determined by the display sequence of the attributes in the View Object. See Determine the Display Sequence of Attributes within a Row

Hide Attributes on the Form Page

With the **Display?** property, you can determine which attributes will be generated in the form page.



This property has three possible values:

1. True. The attribute is generated in the form page
2. False. The attribute is not present in the generated form page
3. Hidden. The attribute is generated in the form page as a hidden form field.

Compared to previous releases of JHeadstart, there is now less need to have hidden fields. With ADF, view and model layer are clearly separated. So set **Display**=false for

attributes you do not want to show. The model layer will still have the values for those non-displayed attributes.

Attention: See the help in the BC Properties editor for more info.

Using regions

By default, JHeadstart places all attributes on the page in the order you have defined them in the View Object.

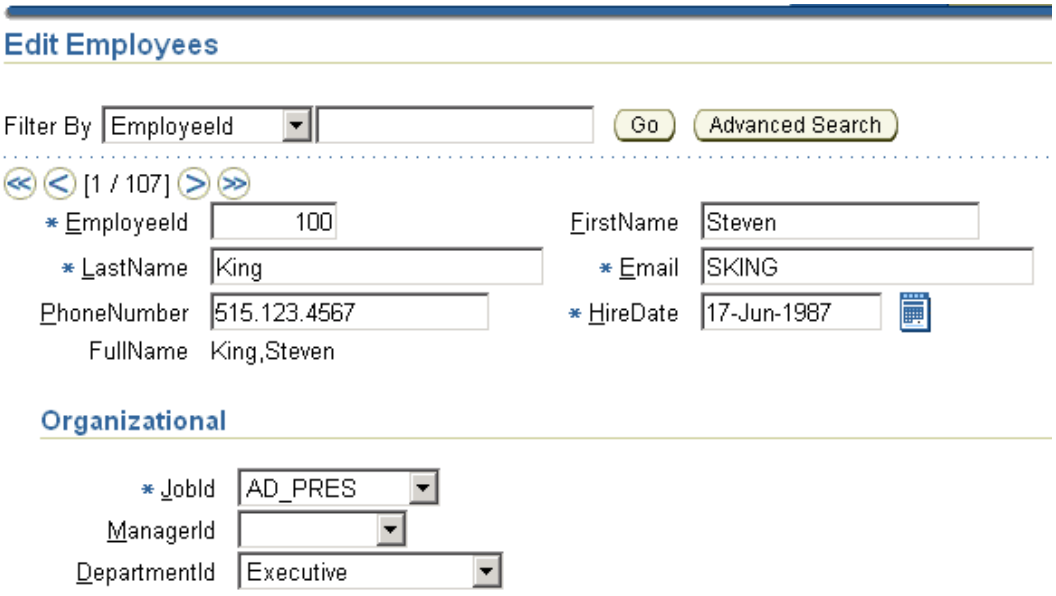
However, often you want to group related attributes of a View Object together. For example: you have attributes with Address information and you have attributes with Financial information.

For this purpose, you can define Regions:

1. Within the group, add the regions you need. See Creating a new Lookup or Region
2. Place the attributes in the appropriate region.

Attention: Regions are only used when generating single-row pages. So only use regions when you have **Layout Style** set to one of: form, table form, select-form, tree-form

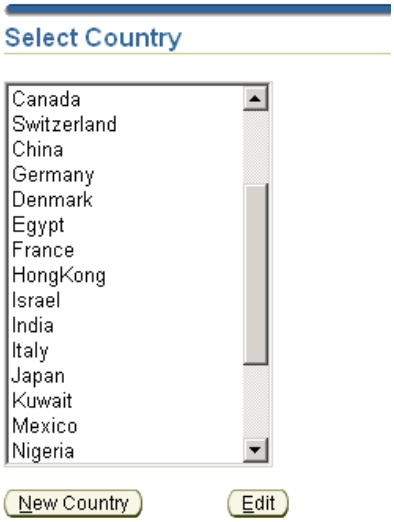
Example of the use of a region. The group level **columns** property is set to 2. The region level **columns** property is set to 1.



Creating Select-Form Pages

A Select-Form layout consists of:

- A Select page with a list box where the user can select a row, and
- A Single Record (Form) page to enter or update a new row.



When Edit is pressed then the form page is displayed. That is similar to the page as shown in the form layout example above.

Use a Select-Form page when the number of records is fairly small.

Add the following attributes to your group definition in the Application Structure File to generate a Select-Form Page:

1. Set the **Layout Style** attribute to 'select-form'.
2. Determine which attribute should be displayed on the Select page if you have not already done so.
   The Select Page displays a list box that contains one unique attribute from the default view object. The user can then find the appropriate record, select it, and perform the desired action (View, Edit, Delete).
   The **Descriptor Attribute** identifies which attribute is to be displayed in the list box. You can only display one view object attribute in the list box, but it could be a transient attribute that contains a concatenation of a number of fields queried from the database. See the section Create Calculated or Transient Attributes on how to create such a transient attribute.
3. See the section 'Creating Form Pages' above which properties are appropriate for the Form page. All properties that are appropriate to the pure Form Page also apply to the Form section of the Select-Form page.
4. Set **Use table range?** to false. Otherwise the list box will show only the first set of records.

Example `'Application Structure File with select-form layout in Group Definition'`

In this example the Countries page should be displayed in a select-form layout.

Query Settings	
* View Object Usage	Countries
* View Object	CountriesView
Group Layout	
* Layout Style	select-form
Search Settings	
Labels	
Tabname	Countries
* Display Title (Plural)	Countries
* Display Title (Singular)	Country
* Descriptor Attribute	CountryName

Creating Table Pages

In many situations you want to present multiple records to the user in one page. For example:

- A page showing all the Countries.
- A page showing all the Employees.

You can generate this type of pages in a number of ways:

1. Using a Table Page. In a Table Page the records are manipulated in the table. Use this option when the number of attributes in the table is small, so the table fits on the page. See the remainder of this section.
2. Using a Table Page with detail disclosure. Use this when the number of attributes is too big to fit on one row. With detail disclosure you can expand/collapse a row in the table. So you can edit one row, while still seeing the other rows. See Show attributes not displayed in the table (UIX only).
3. Using a Table-Form Page. Use this when the number of attributes is too big to fit on one page. This is a combination of a Table Layout with multiple records and a Form Layout for manipulating one record. From the table page you can navigate to a form page to manipulate one record. See Creating Table-Form Pages

Add the following properties to your group definition in the Application Structure File to generate a Table Page:

1. Set the **LayoutStyle** property to table.
2. Determine the amount of horizontal space the table can consume on the page as a percentage (**Table Width**), e.g. 60%. You can also indicate the number of pixels (e.g. 600).

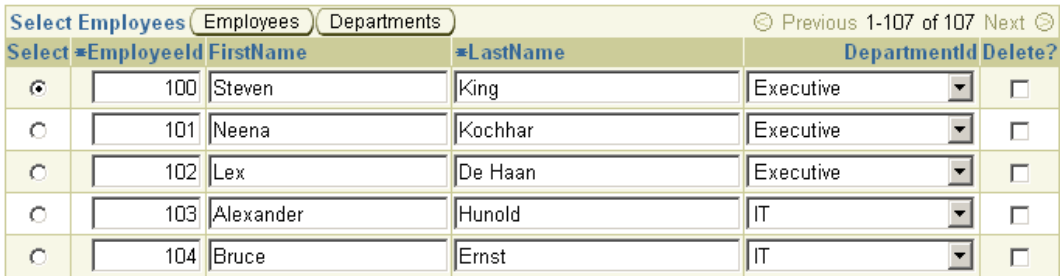
Example `'Application Structure File with table layout in Group Definition'`

In this example the Countries page should be displayed in a table layout. We make the following settings in the Application Structure File Editor:

Group Layout	
* LayoutStyle	table

Table Layout	
Table Width	100%

The generated page looks as follows:



Attention: The order of the rows in the table is determined by the Order By clause in the View Object. See Determine the Order of Displayed Rows

Hide Attributes in a table

With the **Display in Tables?** property, you can determine which attributes will be generated in a table. Values and meaning are the same as for the **Display ?** property.

So when you do not want an attribute to be generated in a table page but you do want to show that attribute in a form page, set **Display in Tables?** to false.

When you do not want to show an attribute at all, set **Display** to false.

Allowing the user to sort data in a table page

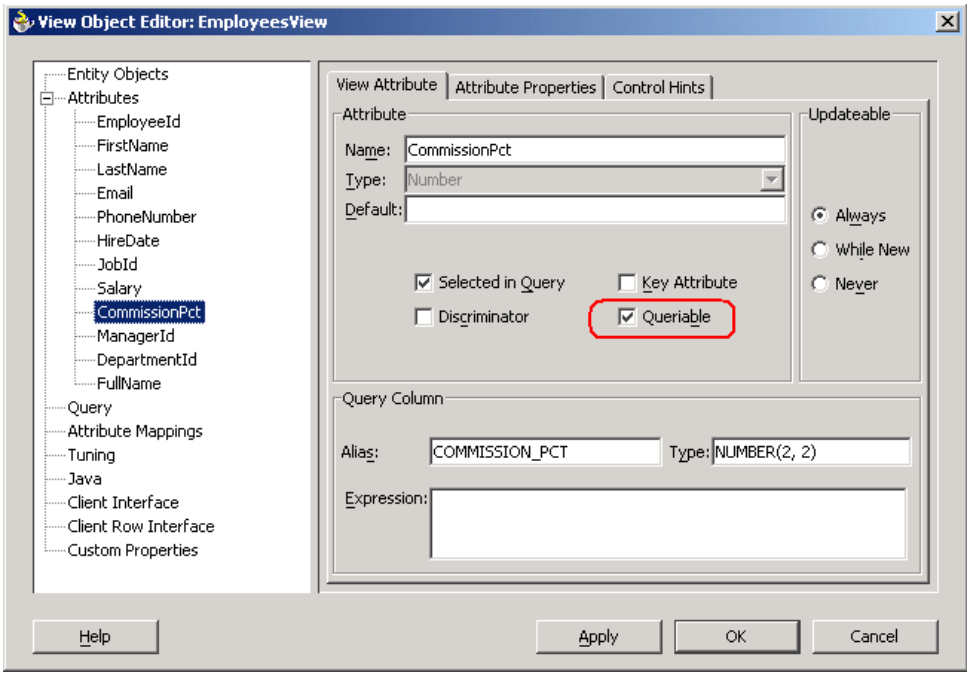
It is possible to generate a feature where the user can do an online sort of the records queried in a table. The user can simply click on the column header and then the table content is sorted based on the values in this column. Clicking the same column header twice will switch the sort order from ascending to descending and vice versa.

If this is required, then you must set the **sortable** property in your group definition to true. All the Attributes in the ViewObject that are queriable will become sortable on the page.

You need to review each View Object and identify attributes that should not logically be sortable. When creating the Entity and View Objects this property is set by default.

Steps for changing the Queriable attribute:

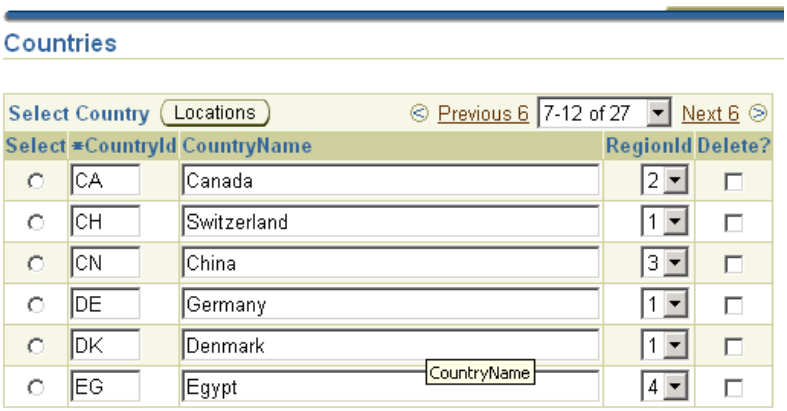
1. Edit the View Object and select the attribute.
2. On the 'View Attribute' tab, check or uncheck the Queriable checkbox.



Limiting the number of records on a table page

By default, the Table and Table-Form layout will display all existing rows in the table. For large tables, this might be undesirable. You can limit the number of records to be displayed at once, and generate a poplist to navigate to another range of rows within the table together with 'next' and 'previous' hyperlinks:

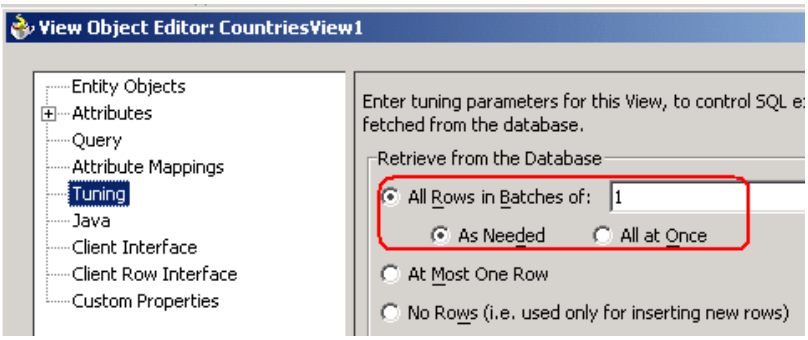
1. Set the **Use Table Range?** property to true for the Group in the Application Structure File editor.
2. Set the **Table range size** property to the number of rows you want to display at once.



This example shows the Countries group with the **useTableRange** property set to true, and the tableRangeSize set to 6. You can now see that a poplist has been generated to select other sets with records, and Previous and Next hyperlinks to navigate to the next or previous record set in the table.

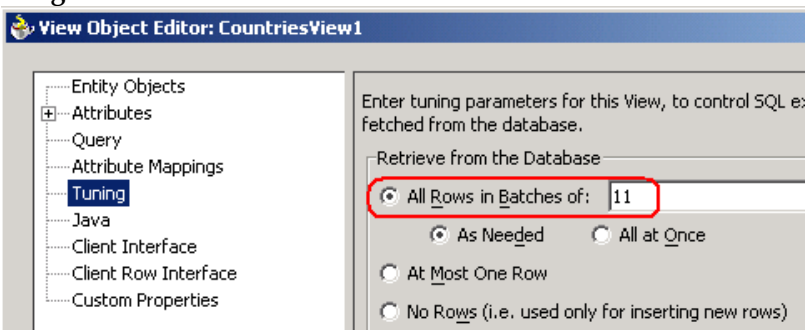
Change related ADF Business Components Settings

By default, ADF BC View Objects fetch rows from the database one at a time. So for each row there is a round-trip from the application server to the database.

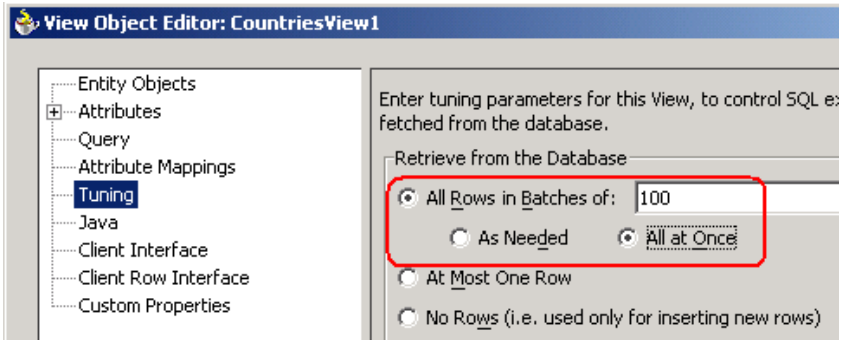


When retrieving multiple rows at a time, this is an unnecessary slow-down. So we recommend adapting the settings in the ViewObject to the settings in the JHeadstart group as follows:

1. When **Use Table Range ?** is true, set All Rows in Batches to the value of **Table Range Size** + 1.



2. When **Layout Style** is table/table-form and **Use Table Range ?** is false, change the View Objects to retrieve all at once. In this case, all records will be shown in the table, so it make sense to fetch all in one call from the database.



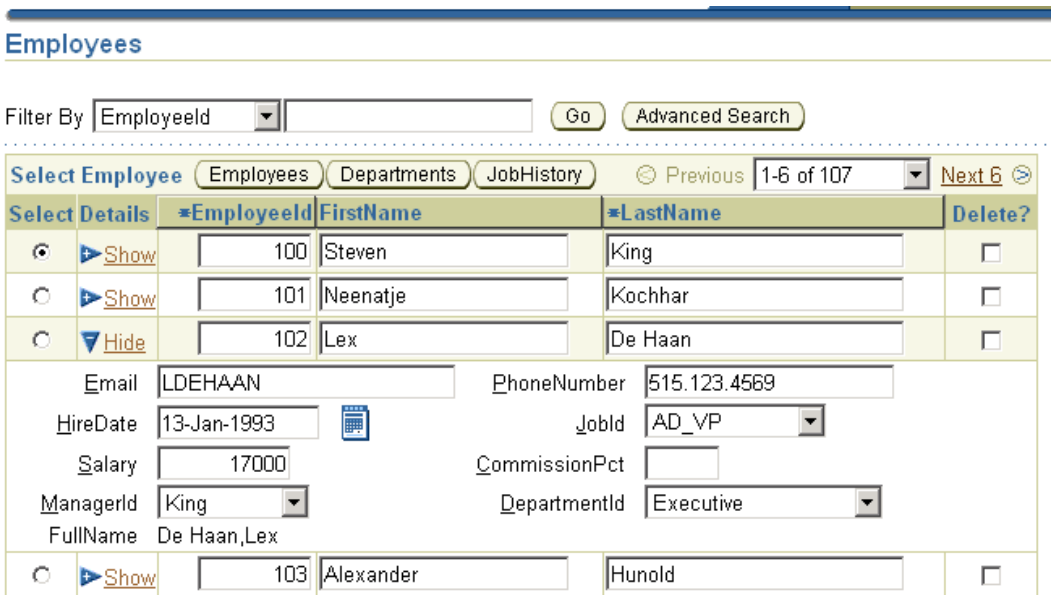
Reference: There is much more to say about ADF Business Components tuning. This sections explains only the ADF BC settings that are directly related to values of JHeadstart properties. For more information on ADF BC tuning, read the tips on Oracle Technet.
For example the article View Object Tuning Tips for Best Performance
http://www.oracle.com/technology/products/jdev/tips/muench/voperftips/index.html

Show attributes not displayed in the table (UIX only)

If you do not display all the attributes in the table you may want to allow the user to see the other attribute values if they want to. Which attributes are displayed are determined based on the **Display in tables** property for each attribute. All attributes for which this property has been set to false are not displayed in the table. However, there might be situations where the end user wants to see these values. This can be achieved by using the **Detail Disclosure** property for the group. When set to true, the table will generate a column in the table with an arrow that allows the end user to disclose details of the current row. The detail section displays all items that have the **Display in Tables?** property set to 'false' and **Display?** set to 'true'.

Show attributes not displayed in the table (UIX only)

If you do not display all the attributes in the table you may want to allow the user to see the other attribute values if they want to. Which attributes are displayed are determined based on the **Display in tables** property for each attribute. All attributes for which this property has been set to false are not displayed in the table. However, there might be situations where the end user wants to see these values. This can be achieved by using the **Detail Disclosure** property for the group. When set to true, the table will generate a column in the table with an arrow that allows the end user to disclose details of the current row. The detail section displays all items that have the **Display in Tables?** property set to 'false' and **Display?** set to 'true'.

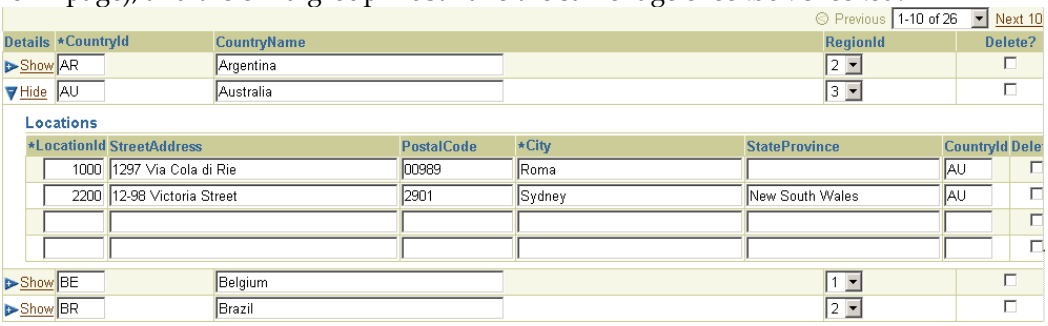


This example shows the Employees group with the **Detail Disclosure** property set to true. You can see that the details are displayed for the third record. The details section has two columns as defined by the **columns** property for the group.

Show nested table (UIX only)

Using the same **Detail Disclosure** property for the group, you can generate child tables within the detail disclosure area. For this to work, the layout style of the group must be set to "table" (with layout style "table-form" the detail table will only be shown on the

form page), and the child group must have the samePage checkbox checked.



Note that the nested table could itself have another nested table, allowing you to nest groups on the same page as many levels deep as you want.

Creating Table-Form Pages

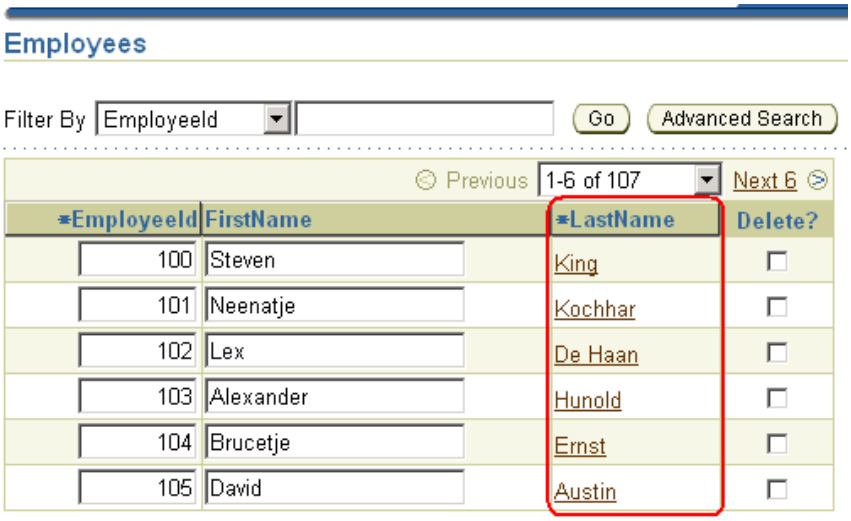
A Table-Form page is a combination of a table format page and a single row page called a form page. In the Table Page the user can update and select a row. If the user selects a record in the Table format page and presses the button to view the details, then the single row page opens, and the user can manipulate or create new rows.

Attention: The Table-Form page layout consists of a combination of the Table page layout and the Form page layout. You can use the group properties described specifically for Table layout to layout the Table part of the Table-Form layout. Also, you can use the group properties described specifically for the Form layout to layout the Form part of the Table-Form layout. View the sections for Table pages and Form pages to see which properties are available.

Steps to create a table-form page

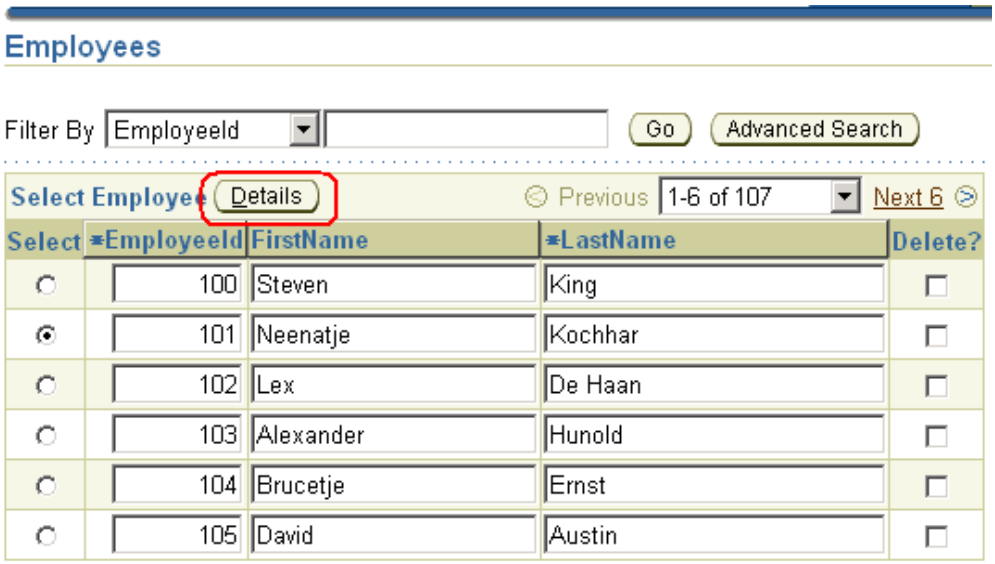
1. Set the **Layout Style** property to 'table-form' to generate a Table-Form Page.
2. Set **Display?** property to false for attributes you do not want to show at all.
3. Set **Display in Tables?** property to true for attributes you want to have in the table page.
4. Set **Display in Tables?** property to false for attributes you want to have in the form page.
5. Choose between a button or hyperlink for means of navigation to the form page by setting the **Table-Form link** property for the group. See the Help in the ASFE.

When you choose a link for navigation to the form page, you will get this:



Suggestion: The link is generated on the descriptor attribute. It is therefore a good idea to set the descriptor to a unique key to help the user distinguish the rows and make the descriptor attribute the first attribute shown in the View Object.

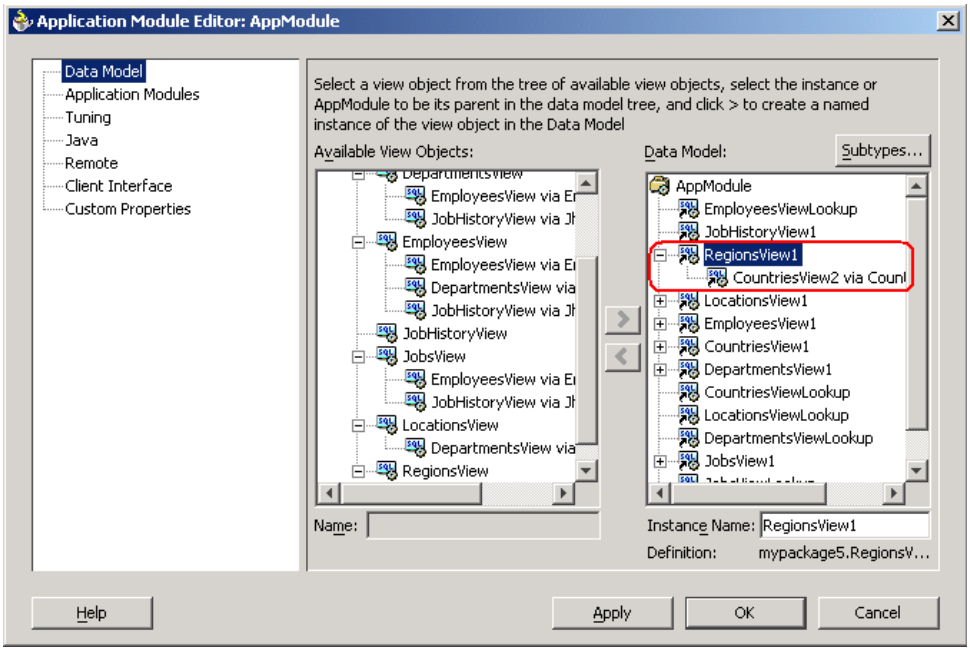
When you choose a button for navigation to the form page, you will get this:



Creating Master-Detail Pages

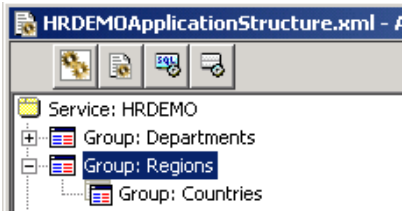
You may want to create pages that are related together as in a master-detail relationship. To be able to generate such master-details on separate pages you should perform the following steps:

1. Check the data model of your Application Module. The master-detail relation should be present as nested View Instances.



Reference: When necessary, correct your Model. See Prepare Model for Generation.

2. Create a group in the Application Structure File for the master page, and create a detail group for the detail page. Set View Object Usage for both groups. Note that for the detail group, you can only select View Objects that are detail View Objects of the master View Object. When the View Object you need does not show up in the Detail Group, go back to step 1 and correct your Data Model.



Master-detail on separate page

1. Set the **samePage** property to false for the detail group. This indicates that the detail should be generated on a separate page.
2. Determine the name of the top level tab for the parent group. Specify this using the **tabName** property as you do for other layouts.
3. Determine the name of the header bar for the level 2 tab bars. Specify this by using the **displayTitle** property for both the master and the detail group.

Example `'Application Structure File Editor with master-detail on separate pages'`

In this example the Region and Country groups are displayed on separate pages.

As you can see two subpages are generated on the Regions tab. The name of the tabs are as defined by the **displayTitle** property. When pressing the Countries tab we get the second page:

As you can see the Countries are shown on the second page. Notice that above the table the name of the region is shown. The information that is shown here is dependent on the **Descriptor Attribute** specified for the master group.

Creating master-detail on same page

To be able to generate master-details on a single page you should perform the following steps:

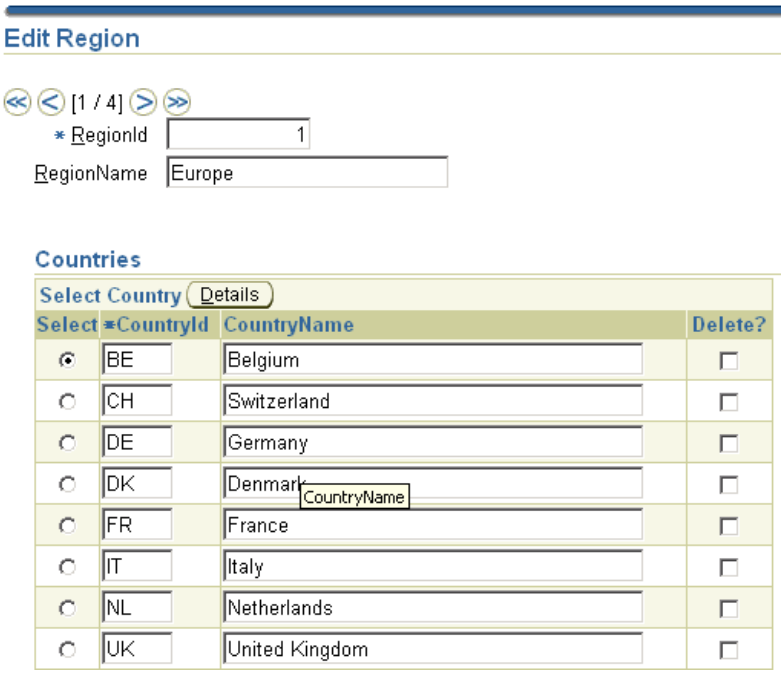
1. Set the **samePage** property to true for the child group. This indicates that the detail should be generated on a separate page.
2. Determine the header of the child group as displayed you want it to be displayed above the child group. Specify this by using the **displayTitle** property for the detail group.

Warning: Out-of-the-box JHeadstart has a maximum of 10 child groups that can be on the same page. You can extend this maximum by modifying the JHeadstart Generator Templates whose name ends with 'PC' (for Parent-Child). For more information, see section 'Using Generator Templates'.

Example `'Application Structure File Editor with master-detail on same pages':`

In this example the Region and the Country groups are defined to be displayed on the same page.

As you can see the master and the detail group have been generated on the same page in a master-detail layout. Note that the header above the details has been set as defined by the **Display Title** property.



Creating Tree Layouts

When your view type is UIX, you can use JHeadstart to generate tree controls. A tree control is extremely useful for showing hierarchical structures in your datamodel.

Examples:

- Geographical areas subdivided in regions.
- Bill of Material structures: parts consisting of sub parts, consisting of sub-sub parts and so on.
- Organizational structures.

This section will explain how to generate such a tree control with JHeadstart.

Generating a Basic Tree

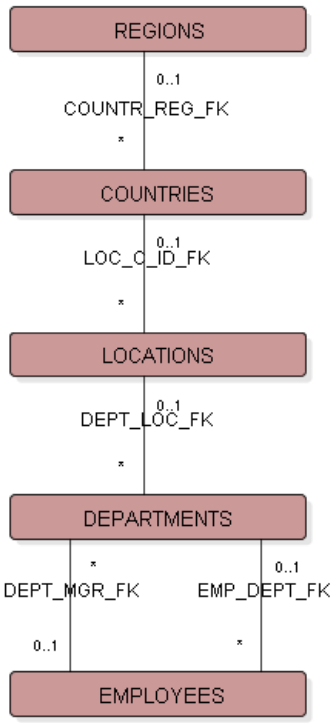
Most of the tree controls you will generate will be of the basic category. There are a few variations that will be explained in later sections:

- Variation: Basic Tree with navigation-only nodes
- Variation: Recursive Tree
- Variation: Recursive Tree with Limited Set of Root Nodes
- Variation: Tree showing only Children of selected Parent

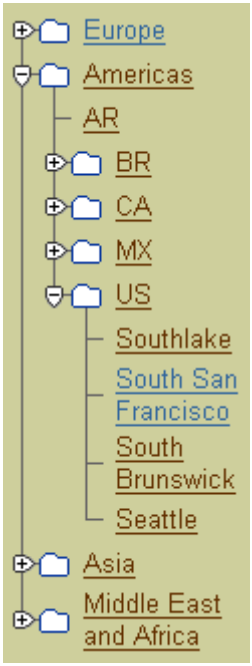
Before reading the variations, start with the basic steps.

In the HR sample schema, a geographical structure is present that can be used in a tree control. We have REGIONS, consisting of multiple COUNTRIES, consisting of multiple LOCATIONS, consisting of multiple DEPARTMENTS, consisting of multiple EMPLOYEES.

So this is our database diagram:



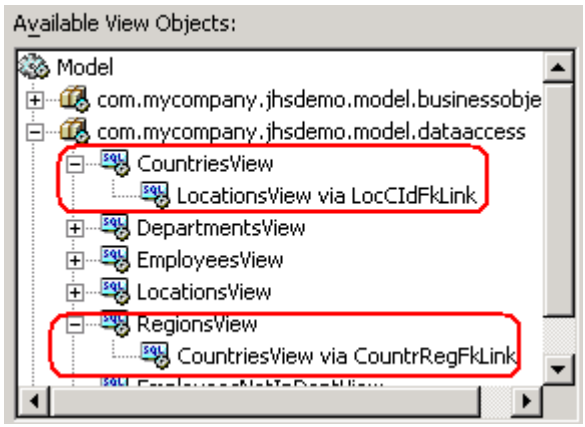
We wil generate a tree with REGIONS, COUNTRIES and LOCATIONS.



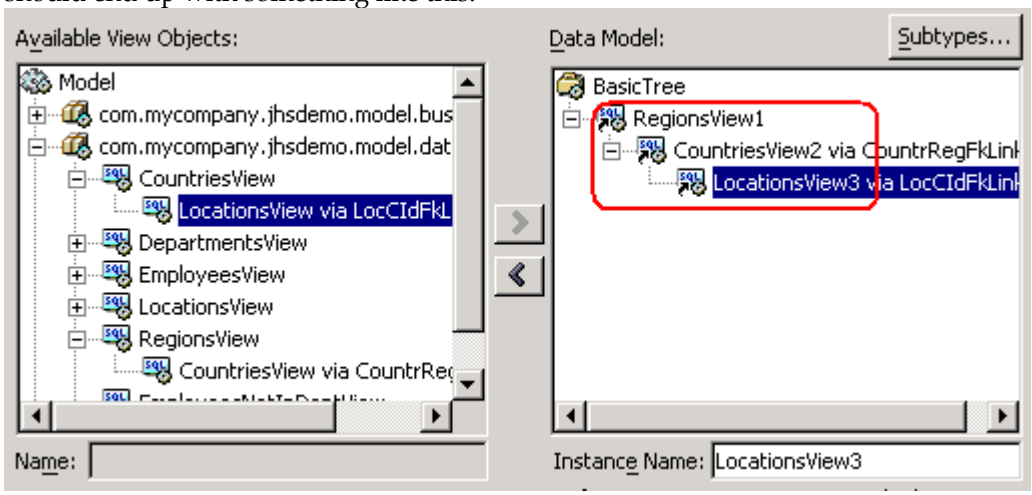
Steps:

1. Check your model. Make sure you have a View Object for each level you want to show in the tree control. Check the presence of View Links between the View Objects. In this example, you will need a View Object for REGIONS, for COUNTRIES

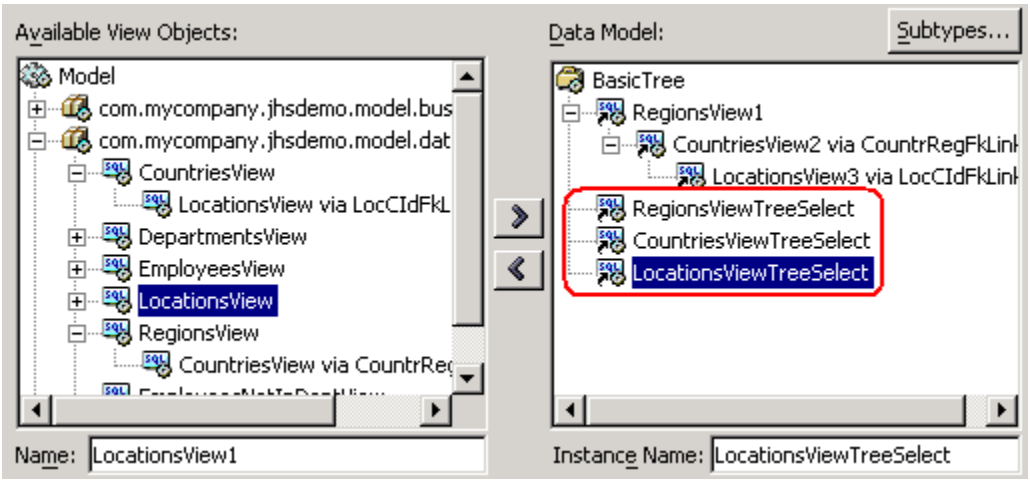
and for LOCATIONS and two View Links for the foreign keys between the tables. You can check this by editing your application module and inspecting the available View Objects. Each View Object should have the correct child View Objects as shown below.



2. Add View Object usages for these View Objects to the Data Model of the Application Module. Add CountriesView as a child of RegionsView and LocationsView as a child of CountriesView. You might need some perseverance here: the user interface of the JDeveloper wizard is not that user-friendly. When adding a detail view, it is important to select the subview in the list of Available View Objects. The subviews are indented in the picture above within the red rectangles. Then select the intended parent View Object usage in the Data Model, and click the right arrow button. You should end up with something like this:



3. For selecting nodes in the tree, you need an extra View Object usage for each type of tree node. It is recommended to create dedicated View Object usages for tree selection, for example RegionsViewTreeSelect, CountriesViewTreeSelect, and LocationsViewTreeSelect. These usages should be top level usages in the Data Model, that is, they should not be a child of another View Object usage.



4. Make sure the JHeadstart Application Structure has groups and detail groups for your tree. You can maintain your groups by hand, or you run the 'New Application Structure File' wizard. Anyway, you should have something like this:

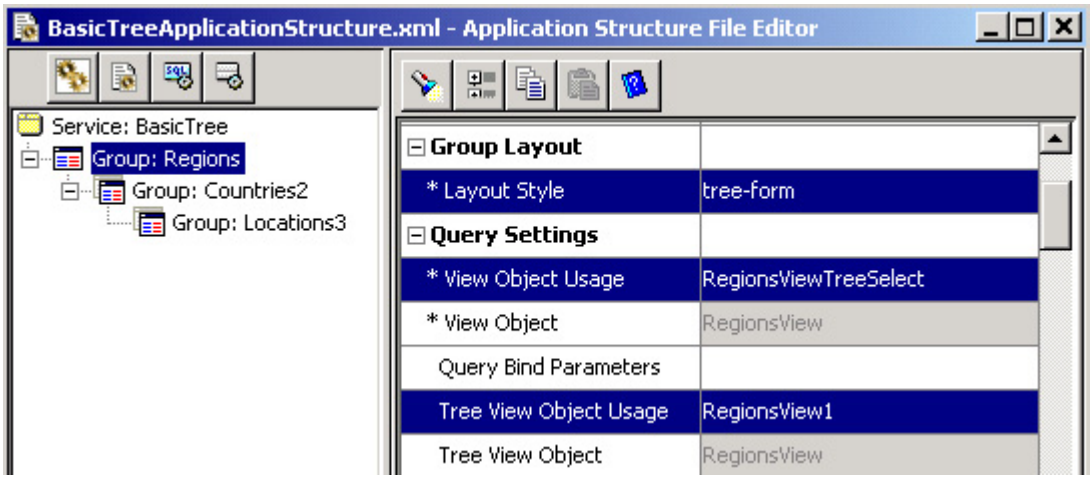
5. For the Regions group and all its detail groups, change the **Layout Style** property to 'tree-form'. The layout style 'tree' will be discussed in one of the next sections.

6. For each of the tree groups, we need to specify two View Object usages:

   The **View Object Usage** property specifies the usage for selecting a tree node and viewing / maintaining the data in a form layout. This is usually a separate TreeSelect usage at top level in the Application Module.

   The **Tree View Object Usage** property specifies the usage for showing the hierarchical structure of the tree, and needs to be a child usage of the parent group (if any).

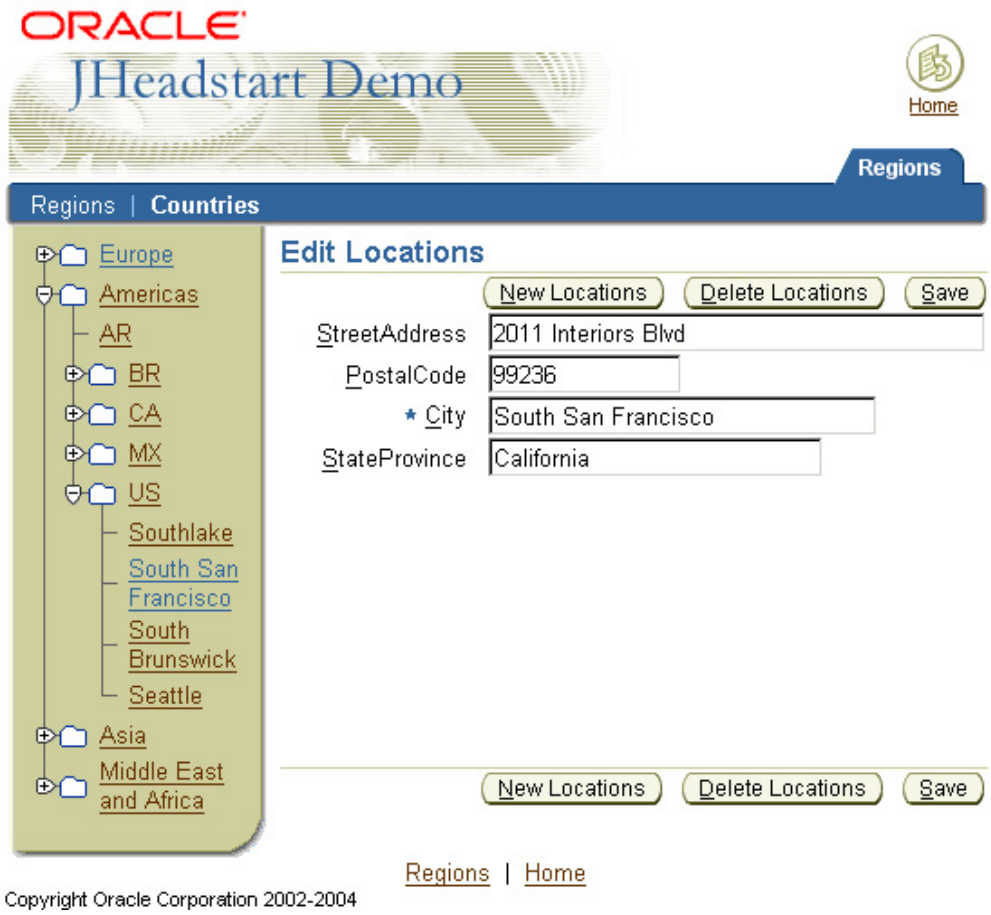
   - If you originally created the groups using the New Application Structure File Wizard, change the **Tree View Object Usage** to be the same as the generated View Object Usage property, and then change the **View Object Usage** property to the TreeSelect usage of that view (for example RegionsViewTreeSelect, CountriesViewTreeSelect, etc).



Attention: The reason why a separate TreeSelect usage is recommended here is the following. If the usage you specify in the View Object Usage property is also used somewhere else in the application, and at the other place a search can be performed on that usage (or in some other way a where clause is applied), you run the risk that the tree selection will not work. It could be that the selected tree node does not occur in the result set that was returned earlier.

7. Select the correct **Descriptor Attribute** for each group to determine which field must be shown in the tree control (for example choose RegionName instead of RegionId). Note: you could also create a new attribute that combines the values of several other attributes, and use that as a descriptor. See Creating a Logical View Descriptor for details.

8. Change the Form Layout properties to your liking and run the JHeadstart Application Generator. You will get something like this:



You can use the tree control to drill down the hierarchical structure.

You can edit records on each level. JHeadstart has added a maintenance page for REGIONS, COUNTRIES and LOCATIONS. You can navigate to the maintenance page by clicking on the appropriate hyperlink in the tree.

Variation: Basic Tree with navigation-only nodes

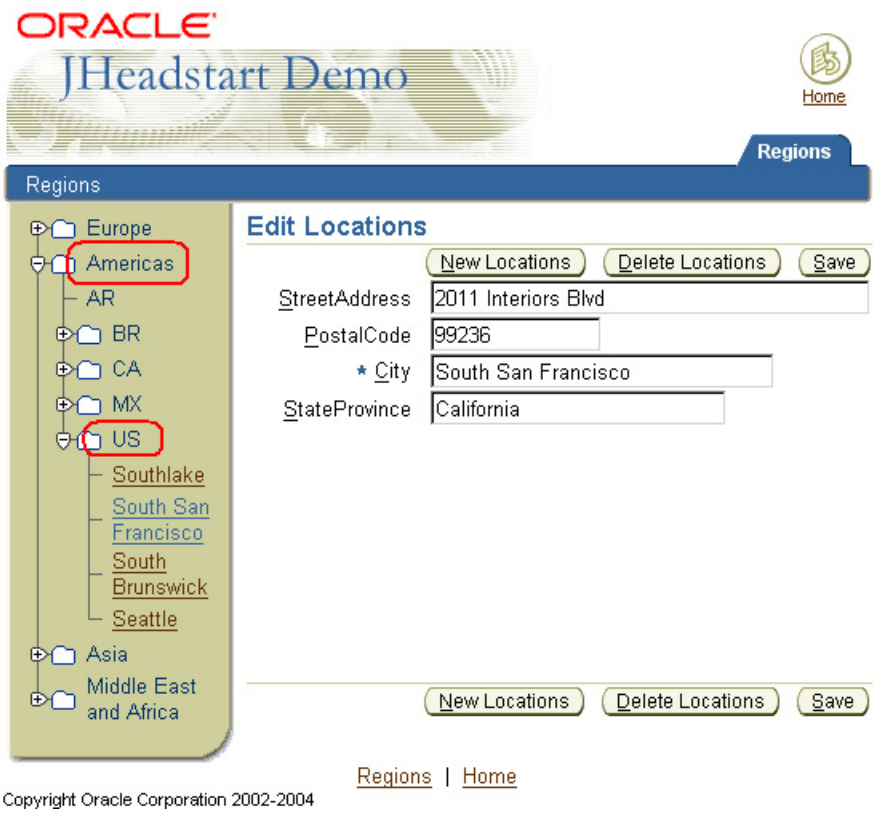
Suppose you do not need editing capability on each level of your tree. For example, you need REGION and COUNTRY only to drill down to the desired LOCATION. In this case, you change the **Layout Style** property of some of the groups to 'tree'. With this layout style, JHeadstart will use the group as a level in the tree control, without generating maintenance pages.

These steps assumed you already applied the steps in section 'Generating a Basic Tree'.

- Change the **Layout Style** property to 'tree' for the groups Regions and Countries2.

☞ **Attention:** The value of the **View Object Usage** (the "tree select usage") property is not used with this layout style, because this tree level cannot be edited. It does not matter which usage you specify, but you must specify one because it is a required property.

- Regenerate. You will get something like this:



Notice the absence of links on the REGIONS ('Americas') and COUNTRIES ('US') level.

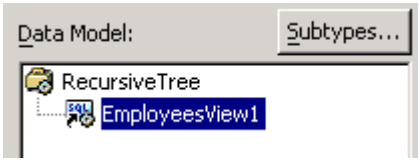
Variation: Recursive Tree

In some cases, tree structures are modeled in the database with self referencing foreign keys (visible in Entity-Relationship diagrams by the so-called pig's ear).

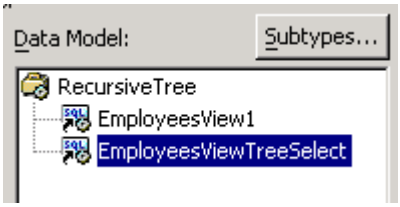
Example: Employees have a manager. The manager is also an employee, so this is modeled as a foreign key from EMPLOYEES to EMPLOYEES.

Generating a tree for such a situation is only slightly different. The steps in section 'Generating a Basic Tree' are applicable to this situation, though with minor changes. The step numbers here are variations of the basic tree steps with the same number.

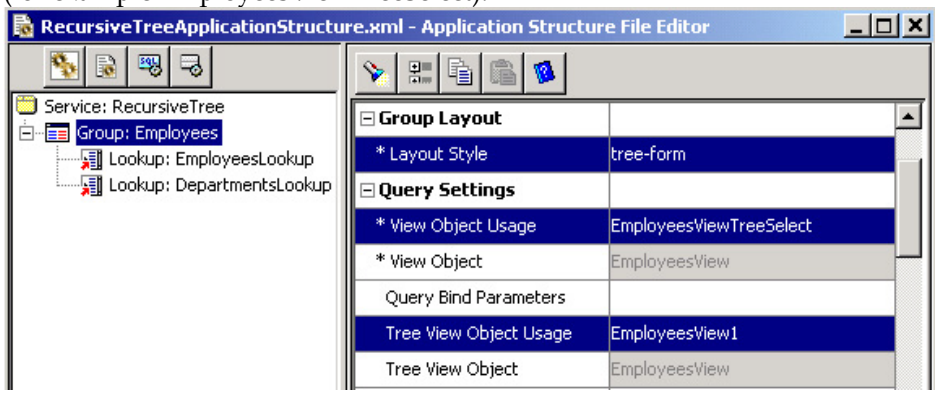
1. You need the self-referencing foreign key as a View Link in your ADF Business Components. The wizard 'New Business Components from Tables' will automatically create such aView Link if a self-referencing foreign key is present in the database.
2. In the data model of the Application Module, it is sufficient to have only one level to generate a tree with an unlimited level of nesting. For example, to have a tree with unlimited recursion on Employees, you only need this data model:



3. For this one-level data model it is not strictly necessary to also create a TreeSelect usage, but it is a good habit and might become necessary when the tree is extended.

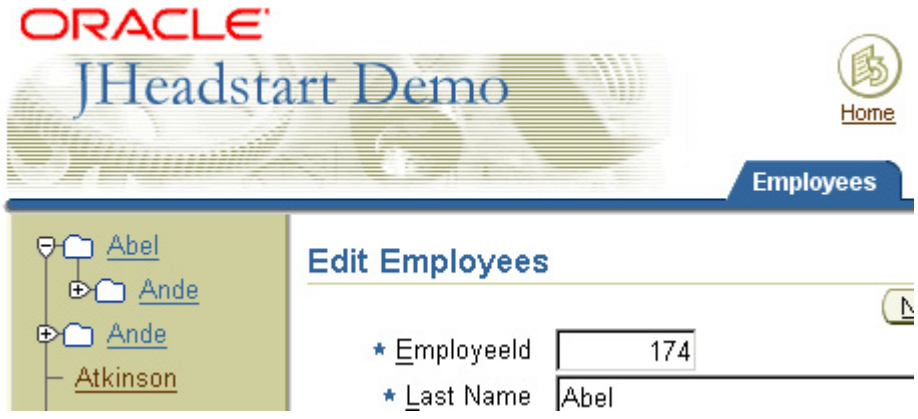


4. You only need one group in your Application Structure file for the self-referencing View Object (similar to the hierarchy in the Application Module data model).
5. Change the **Layout Style** property of this group to 'tree-form'.
6. Change the **Tree View Object Usage** to be the same as the original View Object Usage property (this usage is for showing the hierarchy of tree nodes), and change the **View Object Usage** property to the Tree Selection usage of that view (for example EmployeesViewTreeSelect).



7. Select the right **Descriptor Attribute**.
8. Change the Form Layout properties to your liking, run the JHeadstart Application Generator and you are done.

You will get this (nodes expanded by hand):



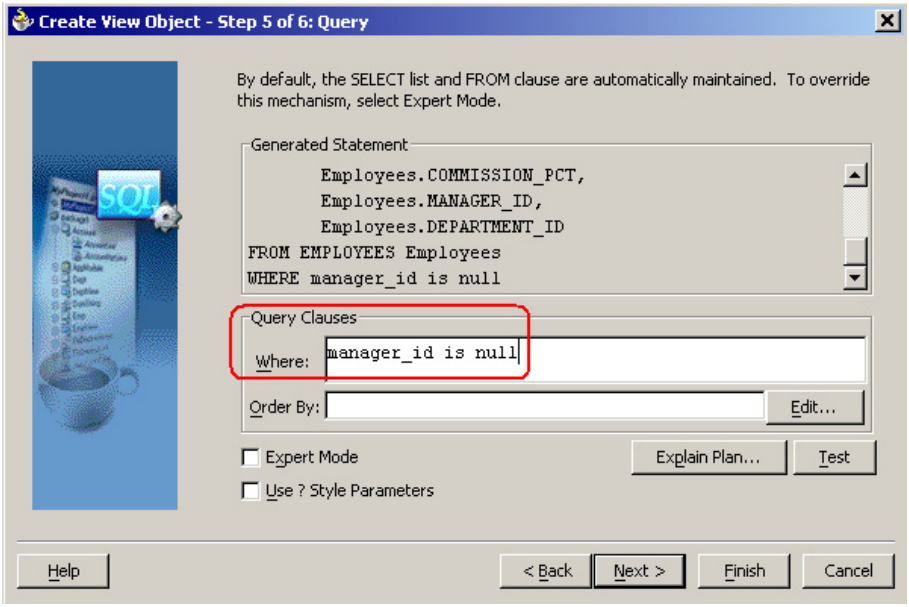
As you see, the tree can expand to any level.

Variation: Recursive Tree with Limited Set of Root Nodes

You can also see in the above tree fragment that employee 'Ande' is displayed twice. By default, every employee is displayed in the top level of the tree. In most cases, this is not what you want. In this example, you most likely want only employees without a manager to appear in the top level of the tree structure, and their subordinates below them. It is quite easy to do so by adding a non-default View Object here.

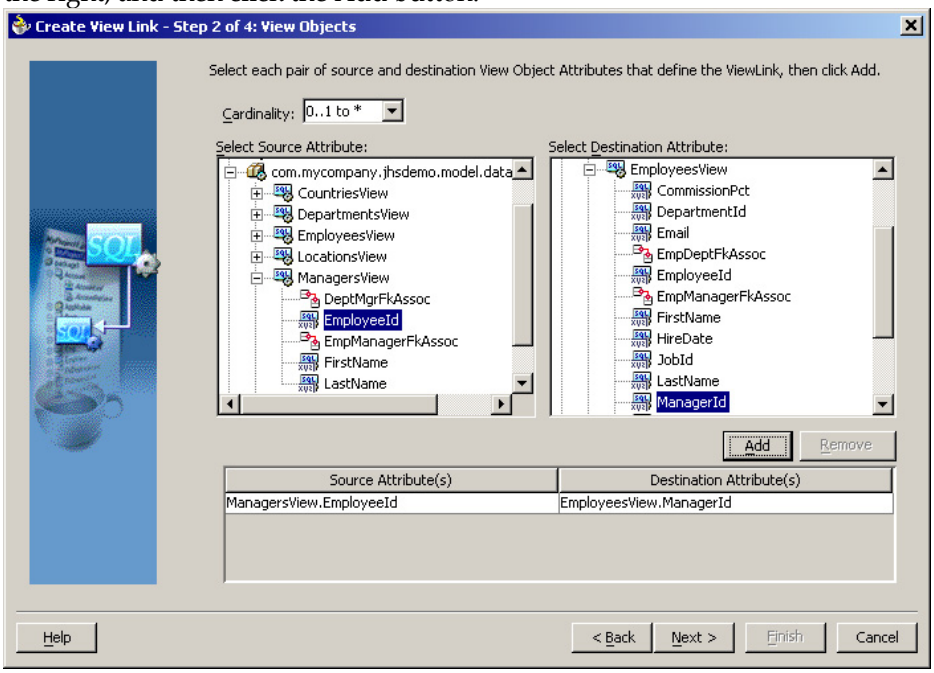
These steps assume you have already done the steps described in section 'Variation: Recursive Tree'.

- Go to your Business Components project and choose 'New View Object'.
- Give the View Object a nice name ('ManagersView') and select the Entity Object and all its attributes.
- Add a Query Where Clause to the View Object. In general, if you want only the root nodes of the recursion, select those rows where the self-referencing foreign key column is null. In this example we select only the employees that have no manager with the Query Where Clause 'manager_id is null'. Test the Query.



- Because this View Object is new, we need also an extra View Link. By doing so, JHeadstart knows the relation between the new View Object with top-level rows and the View Object with the associated subordinates.
- In your Business Components project, choose 'New View Link'.
- Give the View Link an appropriate name, for example MgrHasSubordinatesLink.

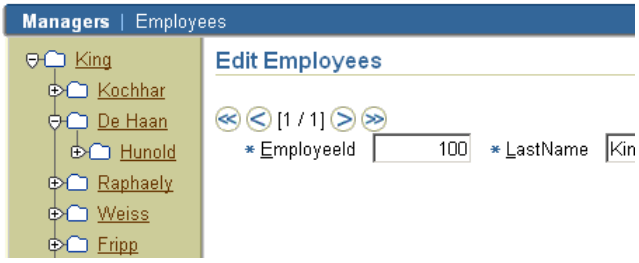
- In step 2 of the 'Create View Link' wizard, select the attributes that relate the data in the two views. In the case of managers and employees, you will select ManagersView.EmployeeId on the left, and EmployeesView.ManagerId on the right, and then click the Add button:



- Add the new View Object 'ManagersView' to the Application Module's data model. Add the detail view with the subordinates 'EmployeesView' below it as a child usage.

Attention: You don't need a new EmployeesViewTreeSelect usage. We can use the one we created earlier, because the attributes are the same as for the ManagersView.

- In the JHeadstart Application Structure File editor, copy the existing Employees group, select the Service node, choose Paste, and call it Managers.
- Cut the original Employees group, select the Managers group, and Paste. This will move the Employees group to be a child of the Managers group. Save, close, and reopen the Application Structure File editor to reset the drop down lists.
- Change the value of the **Tree View Object Usage** property of the Managers group to the new View Object usage 'ManagersView1'. Change the Tree View Object Usage of the child Employees group to the child Employees usage of ManagersView1 (for example 'EmployeesView2'). Leave both View Object Usage properties at 'EmployeesViewTreeSelect'.
- Regenerate. You should get this (nodes in the picture expanded by hand to show contents):



Each record is shown only once in the correct place in the tree structure.

Variation: Tree showing only Children of selected Parent

You might not want to show a tree of all rows in the database, but only of the child nodes of a certain parent row. For example, you want the user first to select a department, and then for that department show a tree of the employees of that department.

With JHeadstart you can generate that by having a hierarchy of groups, and only setting the layout style to tree(-form) for a subset of child groups.

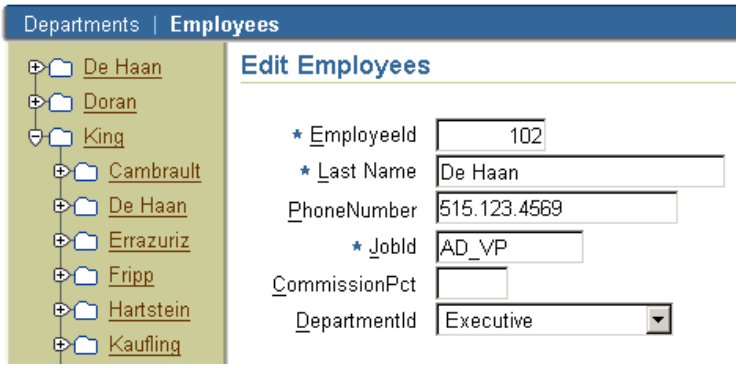
The steps described here assume that you have already built the tree as described in section 'Variation: Recursive Tree'. We cannot use 'Variation: Recursive Tree with Limited Set of Root Nodes' because it would not show any nodes unless the department included a top manager in its employees. You already have a tree of Employees, and now we will add a Department group in front of it.

- First ensure that the Application Module's data model includes the parent-child relations we want to use. We should have a top level DepartmentsView usage, with an EmployeesView child.
- In the Application Structure File Editor, create a new Base Group and call it Departments. Set the **View Object Usage** to the top level DepartmentsView usage. Enter a Tab Name, Display Titles, and a Descriptor Attribute.
- Set Advanced Search to samePage and Quick Search to singleSearchField, on attribute DepartmentName.
- Cut the Employees group of the recursive tree, select the Departments group and Paste. The Employees group has now become a child group of Departments. Save, leave and reopen the Application Structure File Editor to reset the dropdown lists.
- Change the **Tree View Object Usage** of the Employees child group to the child usage of the Departments group.
- Change the **View Object Usage** of the Employees child group from 'EmployeesViewTreeSelect' to the same child usage of Departments.

Warning: The exception to the rule of "Always use a top level View Object usage for tree selection" is when your tree starts at a detail group, and the parent group in your application structure does not have a tree layout. In that case, the first tree group should use a child View Object usage of the parent group for tree selection, to prevent that the default row shown does not belong to the selected parent.

When you run the application, search for a department that has employees (for example 'Human Resources'), and click the Employees subtab. Only then the tree is shown, containing only employees of the selected department, and already a "default" Employee row is selected.

As you can see, the "default" employee (that is shown before any tree node is selected) belongs to the right department. This is because we set the View Object Usage of the Employees child group to the child usage of Departments, instead of using EmployeesViewTreeSelect. If you would have used EmployeesViewTreeSelect, you might see employee 'King' by default, who is not an employee of the Human Resources department.



We are faced with a dilemma when we select a Department like 'Executive'. This department has 'King' as one of its employees, who manages employees that are not in department 'Executive'. When we select Cambrault in the tree, we still see employee 'De Haan' in the Edit page. There is a debug messages saying something like 'No row found in EmployeesView3 with key 148'. Of course, the EmployeesView3 usage now includes only the employees of Human Resources, which does not include Cambrault!

Warning: This issue (selecting a tree node does not work) can only occur when all of the following conditions apply:

1. The tree starts at a child group
2. The tree starts with a recursive (self-referencing) View Object
3. The direct children of the parent might have recursive children that are not a direct child of the parent.

Suggestion: You can solve this by changing the View Object Usage of the highest tree group. The View Object usage you specify should contain every possible selectable tree node of this View Object, including the drill-down nodes. If you cannot implement this using a child usage of the parent group, then there is a workaround: temporarily change the layout style of the parent group to tree, then pick any View Object Usage you want for the child group (for example 'EmployeesViewTreeSelect'), and change back the layout style of the parent. This reintroduces the risk of showing the wrong initial row, however (but that might be preferable over not being able to select some of the tree nodes).

Showing tree initially in expanded mode

By default, the tree is rendered in collapsed node when it is accessed for the first time. If you want to show the tree in expanded mode by default, you can achieve this by suffixing the methodName of the TreeProxy data provider in the treeTemplate.jut file with the word "Expanded":

```
<data name="TreeProxy">
  <method class="oracle.jheadstart.view.adfuix.TreeUtils"
          method="getTreeProxyExpanded"/>
</data>
```

See section Customizing Page Layout Generation for more information on template customization.

Keep tree expanded after saving changes

By default, the tree is rendered in collapsed state again after you have saved the changes in a form page selected through the tree. The reason for this is that you might have updated the foreign key attribute that refers to the parent tree node, which means the current tree state is no longer valid as the child node must be re-parented. However, if your form page does not allow for updating the parent foreign key attribute, you can configure the tree to not collapse on update. You can do this by changing the value of the hidden form field "collapseTreeOnUpdate" from true to false in treeDataPage.jut and treeDataPagePC.jut:

```
<formValue name="collapseTreeOnUpdate" value="false"/>
```

See section Customizing Page Layout Generation for more information on template customization.

Creating Shuttle Layouts

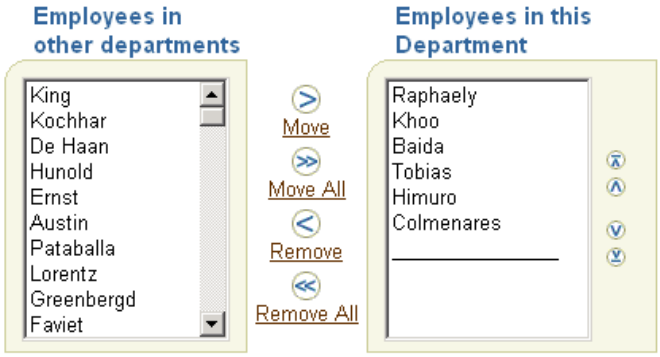
When your view type is UIX, you can use JHeadstart to generate Shuttles.

A shuttle is used to present a list of records to the user. The user can move records from the selected list to unselected and vice versa.

Examples of the use of a shuttle:

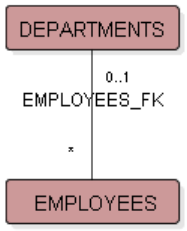
1. Defining employees as members of a department. The left part of the shuttle shows all employees in other departments. The right hand shows the employees that are selected as members of this department. See screenshot below. (JHeadstart calls this a parent-shuttle).

2. Attaching roles to a user. The left hand of the shuttle shows all the roles not attached to the user currently. The right hand shows all the roles the user has already. (JHeadstart calls this a intersection-shuttle)



Creating Parent Shuttles

Use a parent shuttle when you want to attach existing detail records to parents. For example, you want to attach employees to departments, or customers to sales representatives. A parent-shuttle does not create new records, but only updates links to parent records.



With a parent shuttle you can maintain the relation between employees and departments.

In this example we will create a parent shuttle to assign employees to departments.

Steps to create a parent shuttle:

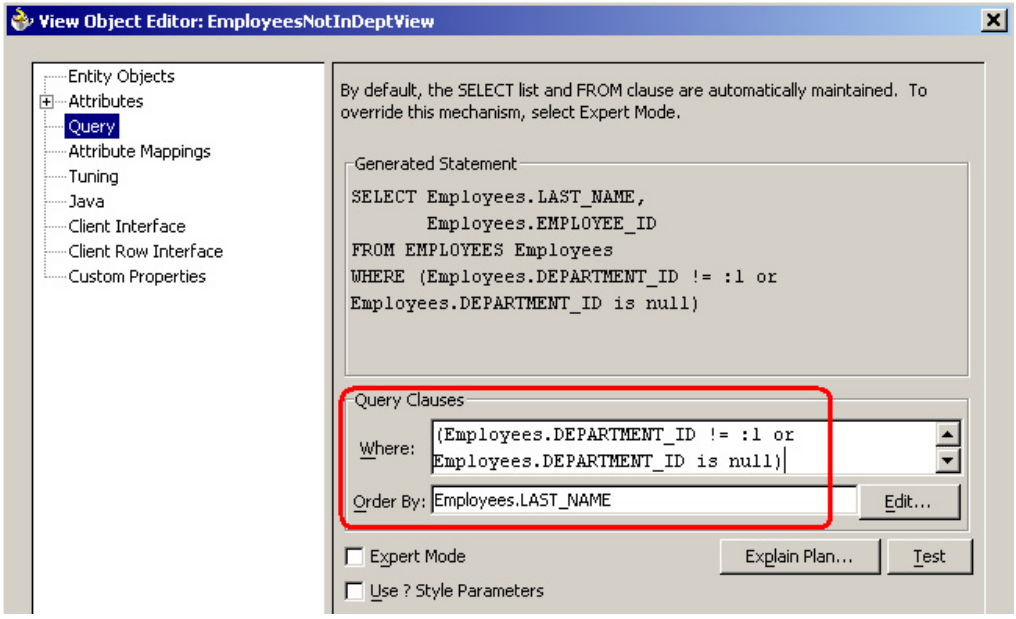
1. Define a base group on the DepartmentsView. Set **Layout Style** to 'form'.

2. Define a detail group on the EmployeesView detail group. The detail group will be used for displaying the 'selected' part of the shuttle. Set **Layout Style** to 'parent-shuttle'. Set **TabName** to 'Assign Employees to Departments. Set **Display Title (plural)** to 'Employees in this Department'

3. Go to your Model project and create a New Default View Object for the Employees Entity Object. Name the new View Object something like EmployeesNotInDeptView. Add the New View Object to your Application Module.

Attention: This new View Object must be based on the same Entity Object as the View Object of step 2, because the employee is transferred to the selected parent department by updating this View Object.

4. Modify the View Object to retrieve only the employees not in this Department, and set the Order By clause.

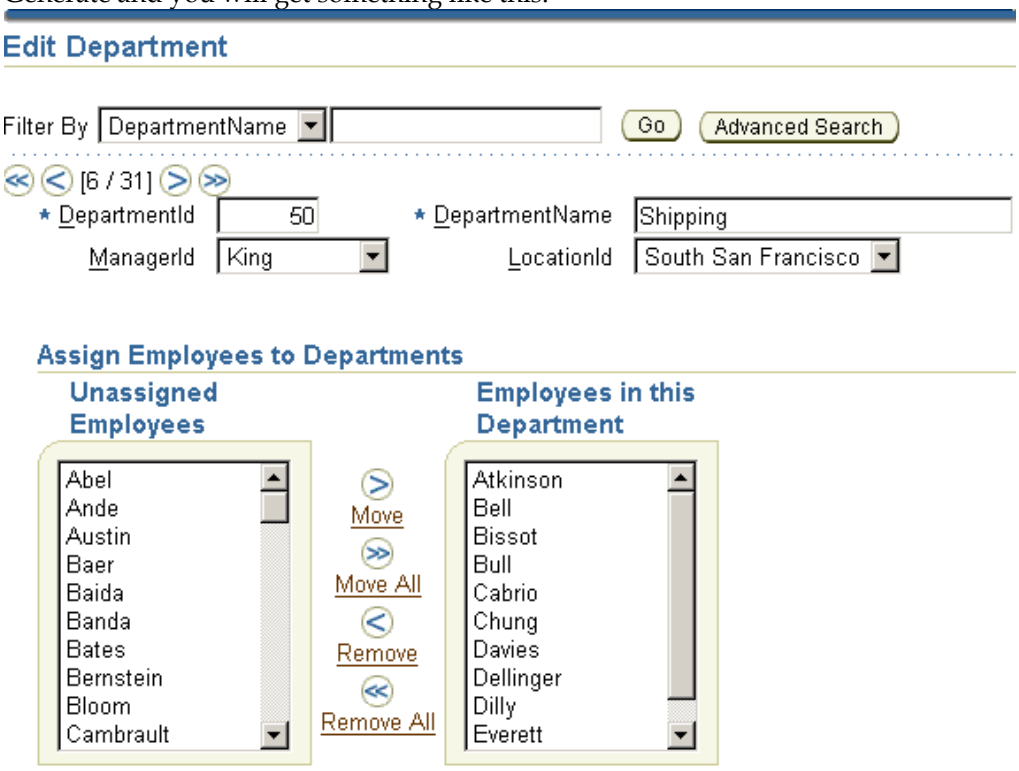


Attention: Note the brackets around the Where Clause: these are needed if you want to be able to perform a search on this View Object. The search adds an AND-clause, which should be added to the whole of this where clause (and not only to the last part of the OR).

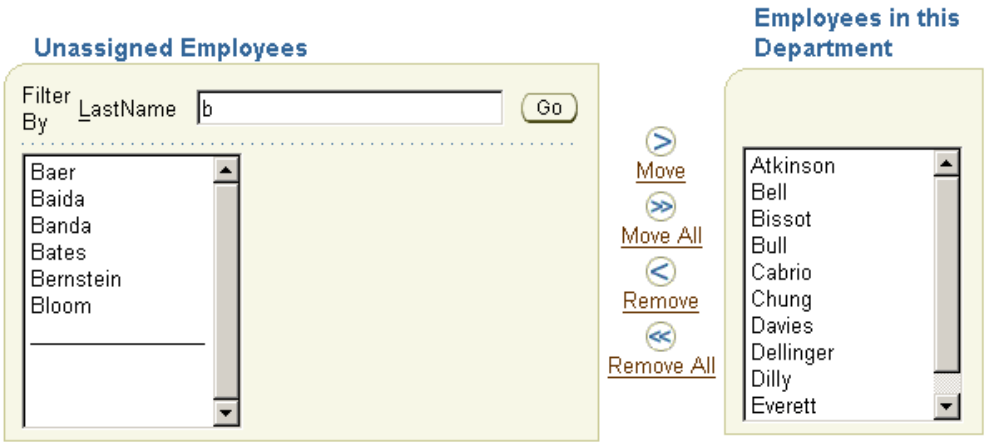
5. Create a new lookup for the child group based on the new View Object. In this case you could update the existing lookup for the Employee Manager, because it won't be used for the shuttle. Make sure the **View Object Usage** property is set to the new view. Disable Advanced Search for the lookup. Set **title** property to 'Unassigned Employees'. Make sure that the Lookup Value Attribute is set to EmpId, the Lookup Display Attribute to LastName, the Base Value Attribute and the Base Display Attribute to EmpId (though the Base Attributes are not used in the case of a parent shuttle, JHeadstart will use the Key attribute of Employees to find the row that needs to be updated).

6. Set property **Query Bind Parameters** to ${data.DepartmentsUIModel.DepartmentsDepartmentId}. When the parent group and shuttle are in the same page, you can also use the short hand EL expression ${bindings. DepartmentsDepartmentId}. ADF uses the key 'bindings' to refer to anything that is in the current UI model (binding container). ADF uses the key 'data' to refer to anything that is in the current binding context.

7. Generate and you will get something like this:



8. You can add a Quick Search Region to the left hand side of the shuttle. Then the shuttle will look something like this:



Attention: When deattaching a record (moving from right to left), the employee has no relation with any department anymore. In database terms, the department_id column is set to null. This means it is best to use a parent-shuttle with an optional foreign key.

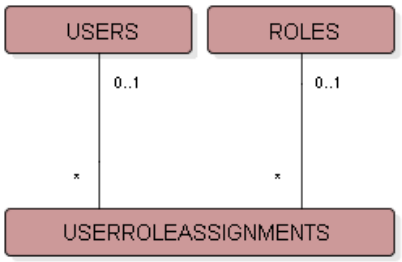
Suggestion: If you want to use a shuttle with a mandatory relation, then you can change the shuttle Generator Template to show no rows in the right-hand side (so that there is nothing to remove). To do this, edit shuttleGroup.jut and/or shuttleChildGroup.jut, and remove the `<contents>` tag from the `<trailing>` tag (keep an empty `<list>` tag). For more information, see section Using Generator Templates.

You could still show the currently attached children in another (non-shuttle) child group of the same base group that is based on the same view as the shuttle group.

Creating Intersection Shuttles

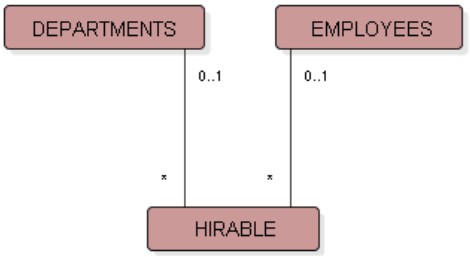
Use an intersection shuttle when you want to mainain an intersection between two ViewObjects. An intersection typically exists when there is a m:n relation between two View Objects. Examples:

- An m:n relation exists between Users and their Roles.
- An m:n relation exists between Employees and Projects.



In such cases, you will most likely implement the m:n relation with an intersection table: a table with two foreign keys to the related tables. With an intersection shuttle, you can maintain the contents of the intersection table.

Because the HR schema does not have a pure intersection table, we will add one:




```
create table hirable (id number, employee_id number, department_id
number);

alter table hirable add constraint hir_pk primary key (id);

alter table hirable add constraint hirdeptfk foreign key
(department_id) references departments;
```

```
alter table hirable add constraint hirempfk foreign key
(employee_id) references employees;
```

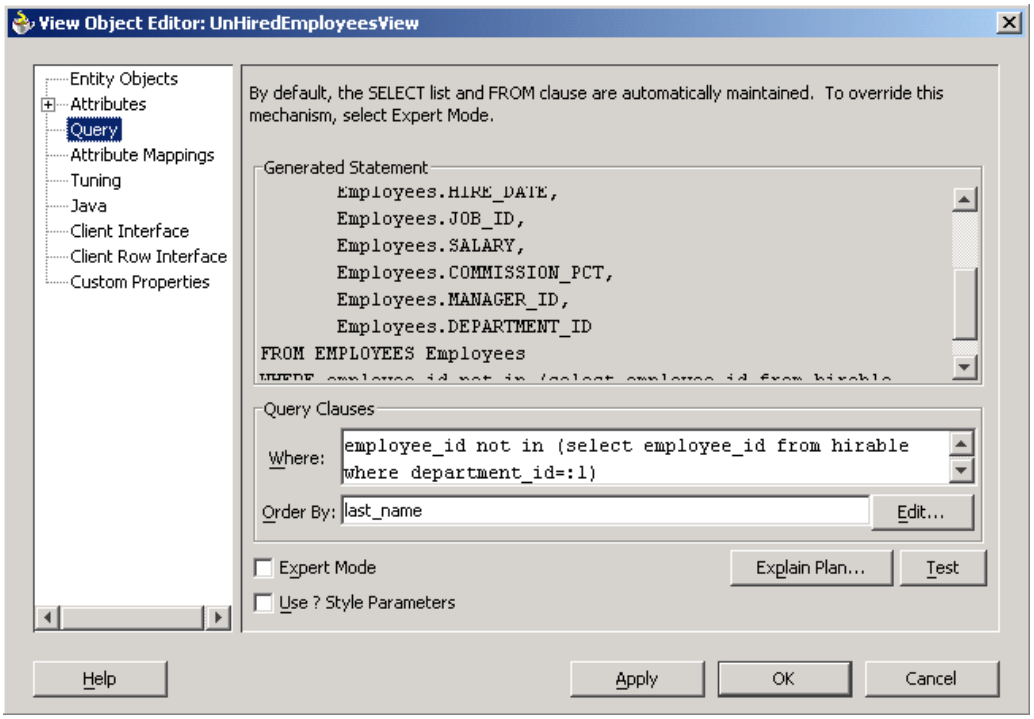
The hirable table is an intersection between Employees and Departments. It relates multiple employees to multiple departments. Each department can hire multiple employees. Each employee is hirable by multiple departments.

Generate Business Components for this new table. You will need an Entity Object, Associations, a Default View Object and View Links.

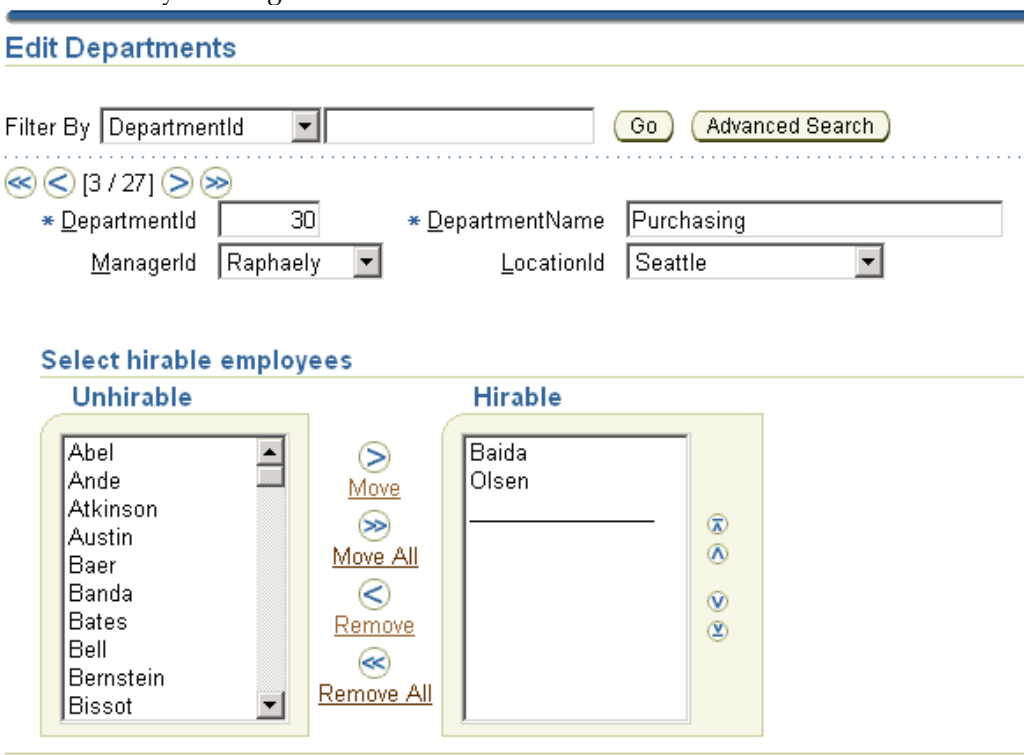
Suggestion: It is easiest to start with a new Model project and regenerate all your business components from start. Then you will get all necessary Associations and View Links.

Steps to create an Intersection Shuttle:

1. Because an Intersection Shuttle will generate new records, have a system in place to generate primary key values for the intersection table. See Generated Primary Key Values.
2. Extend the Hirable View Object with the LastName attribute from the Employees. We need LastName as Descriptor attribute to show in the shuttle. See Add attributes from other Entities to a View object
3. Define a base group on the DepartmentsView. Set **Layout Style** to 'form'.
4. Define a detail group on the Hirable View. The detail group will be used for displaying the 'selected' part of the shuttle. Set **Layout Style** to 'intersection-shuttle'. Set **TabName** to 'Select hirable Employees'. Set **Display Title (plural)** to 'Hirable'. Select LastName as **Descriptor Attribute.**
5. Go to your Model project and create a New Default View Object for the Employees Entity Object. Name the new ViewObject something like UnHiredEmployeesView. Add the new View Object to your Application Module.
6. Modify the UnhiredEmployeesView to retrieve only the employees not hirable by a Department:



7. Base the first lookup of employees on the new View Object created by setting the **View Object Usage** property. Disable Advanced Search for the lookup. Set **title** property to 'Unhirable'.
8. Set property **Query Bind Parameters** to ${bindings.DepartmentsDepartmentId}
9. Generate and you will get this:



Query Behaviour

This section describes how you can influence the query behaviour of generated pages.

Specifying Auto Query

By default, JHeadstart generated pages with Auto Query on. This means that the records are automatically retrieved when the user enters a page, potentially retrieving a large result.

On both the Group and Lookup, you can set the **Auto Query** property to false. This means that records are queried upon request from the user, either by doing a Quick Search or an Advanced Search. This is particular useful when we want to force the user to restrict the number of rows retrieved by specifying search criteria.

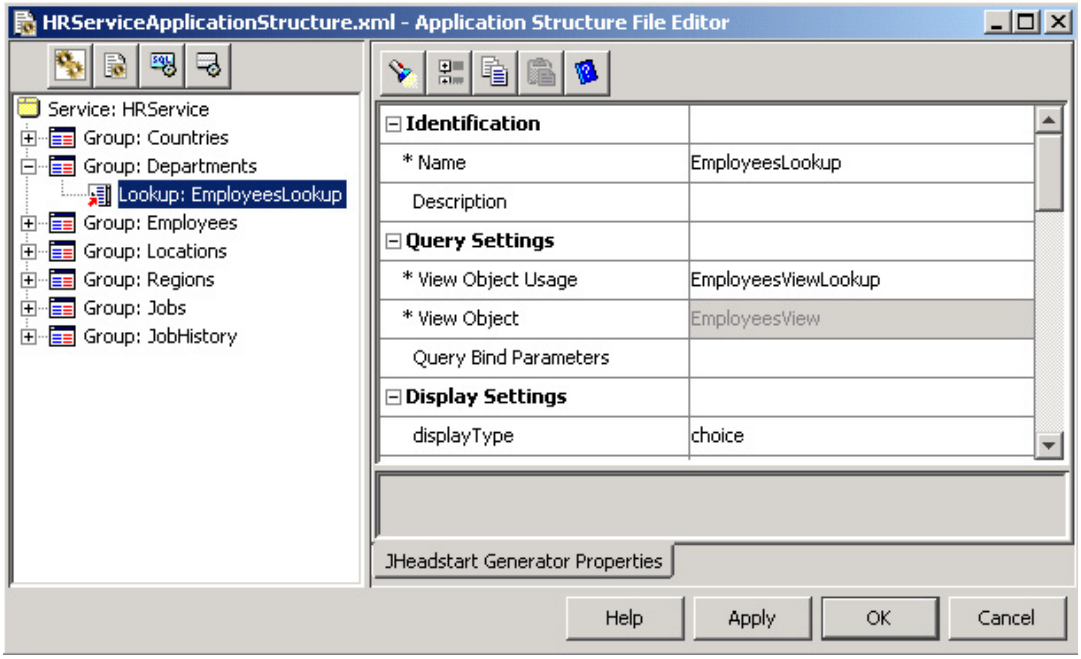
See also the **Maximum Number of Search Hits** property. Use this property to force the user to enter more restrictive search criteria.

Using Query Bind Parameters

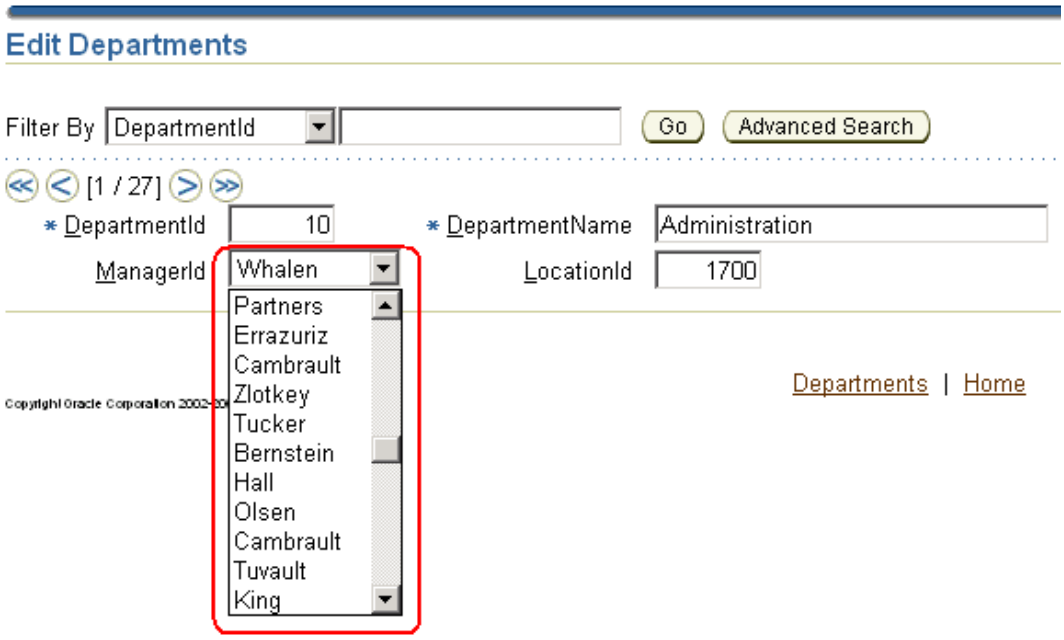
Both Groups and Lookups are based on an ADF View Object. A View Object contains a SQL query. By default, this is a fixed query. This means the View Object will always return the same set of rows with each execution (if the database has not changed). In many cases you want your View Object to be dynamic. For example a View Object that retrieves the Employees of a Department. You want to pass the DepartmentId into the ViewObject and have the ViewObject return the correct rows.

ADF BC View Objects have bind variables for this functionality. JHeadstart can at runtime pass values into these bind variables using the **Query Bind Parameters** property

We will use the example of departments that have a managing employee:



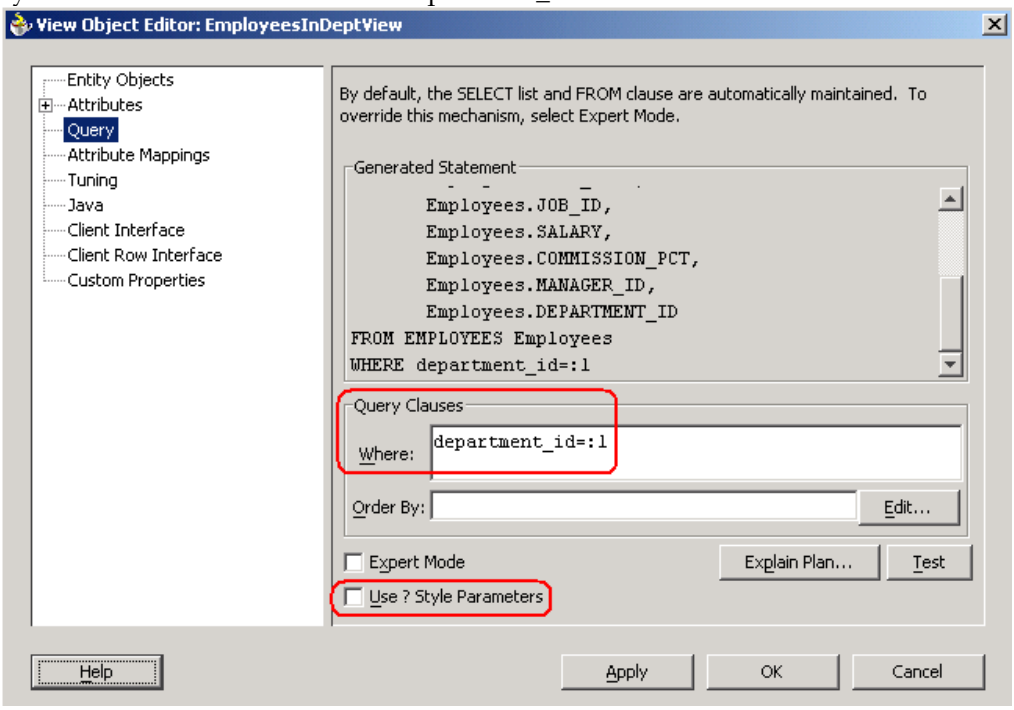
Base group is departments with **Layout Style**='form'. The **displayType** of the Lookup is 'choice'. JHeadstart generates this for us:



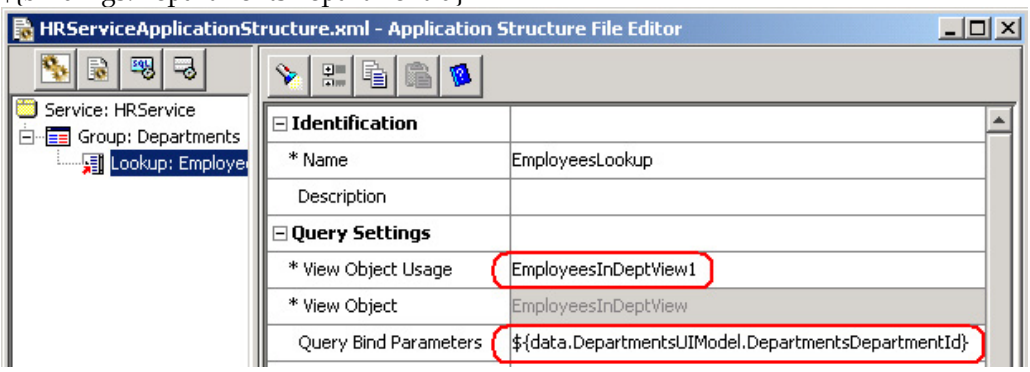
In the dropdown list, all the employees are shown. This is not what we want. We want the manager to be an employee of the department. The dropdown list should only contain employees that are in the department we are maintaining, in this case the department with DepartmentId=10.

We will implement this requirement by using the Query Bind Parameters of JHeadstart:

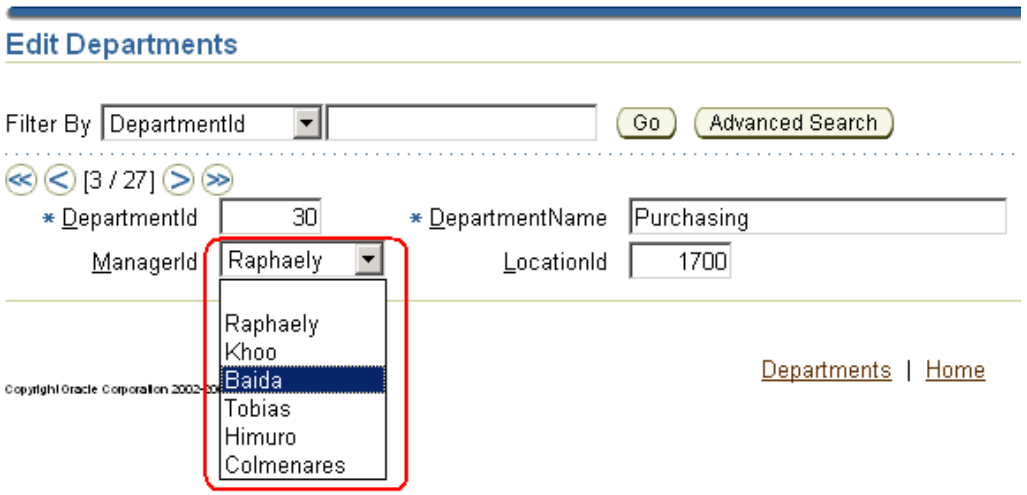
1. Go to your Model project and create a New Default View Object for the Employees Entity Object. Name the new View Object something like 'EmployeesInDeptView'.
2. Edit the New View Object and enter a Query Where Clause with a bind variable. It is important NOT to use the '? Style' parameters. Enter bind variables with ':n' syntax. In this case we will enter 'department_id=:1'



3. In JHeadstart, base the Lookup on the new View Object and enter in the Query Bind Parameters property this EL expression: ${bindings.DepartmentsDepartmentId}



4. Read the Help for the **Query Bind Parameters** property in the ASFE for the use of EL in combination with JHeadstart. In this case we specify that at runtime the value of the DepartmentId is passed to the View Object as value for the ':1' bind parameter. When you have multiple bind parameters in the view (':1', ':2' and so on), you have to specify multiple values in the **Query Bind Parameters** property, comma separated.
5. Generate and run the application again. The dropdown list for ManagerId now contains only employees of the same department.



You can use Query Bind Parameters for both Groups and Lookups. Using EL, you can bind to any value available on the request or the session. JHeadstart will automatically requery when the value of a bind parameter has changed.

Reference: For more info about ADF, UIModel and EL, see Oracle Technology Network. You could start with 'Quick Overview of JSTL, EL and how it's used in Oracle ADF'
http://www.oracle.com/technology/products/jdev/collateral/papers/10g/jstl_el_adf/jstl_el_adf.html

Creating a Search Region

In most cases, you want to give the end-user some query functionality to search for rows with specific values and reduce the number of rows. This section describes how to do that.

JHeadstart is able to generate two distinct ways of search functionality:

1. Quick Search: The search region is placed on top of the generated page. Typically you can only search on one field at a time. Range queries are not supported with quick search.
2. Advanced Search. The search region can be on top of the page or in a separate page. The user can search on multiple fields together. Range queries are supported.

Suggestion: You can use both options together. For example: a Quick Search for the most frequently used selection, and an Advanced Search for less frequently used selections. In that case the Quick Search will be shown by default, with a button to go to the Advanced Search region.

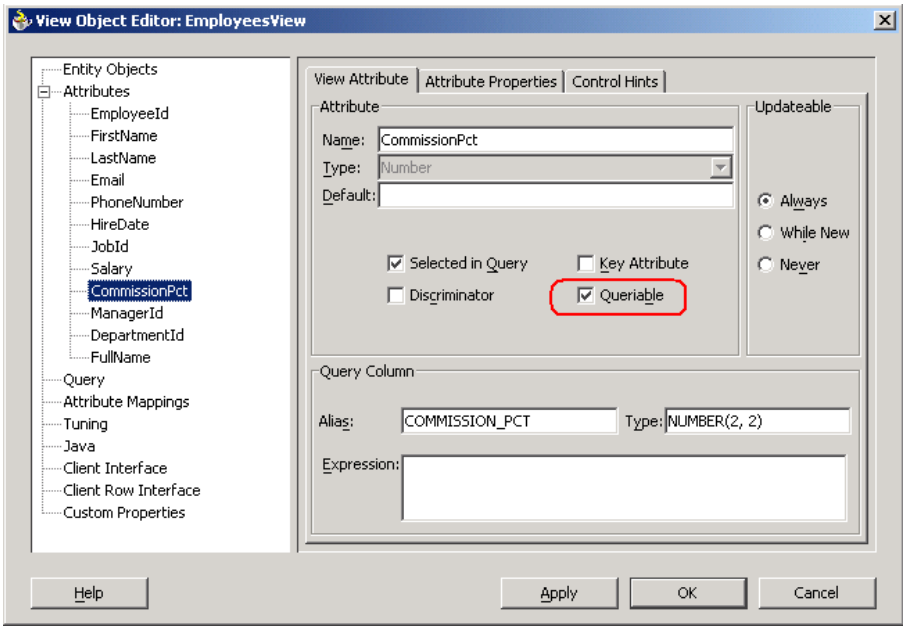
Before generating a Quick Search or Advanced Search page, you have to make some preparations:

Determine which attributes should be displayed in the Search Region (Queriable Attributes)

You need to review each View Object and identify attributes that should not logically be queriable.

If requested to generate search functionality, the JHeadstart Application Generator needs to know what the queriable attributes are. You can set the queriable property for each attribute in a View Object. When creating the Entity and View Objects this property is set by default. Therefore to prevent an item from being included on the Search Region, just edit the item and uncheck Queriable.

1. Right click on the Attribute in the View Object. Choose Edit <attribute>.
2. On the 'View Attribute' tab, uncheck the Queriable checkbox.



Using Quick Search

To generate a Quick Search region for a group, you have two choices:

1. The attribute used for searching is always the same. Give the **Quick Search?** property the value 'Single Field'. Select the search attribute in the **Single Search Attribute** property.
2. You want the user to be able to select the attribute to search on. Set the **Quick Search?** property to 'dropdownList'. JHeadstart will populate a poplist with attribute names so the user can select the attribute to query on. Only queriable attributes are shown in the poplist.

You can also completely disable Quick Search by setting **Quick Search?** to 'none'.

Using Advanced Search

Again, there are two possibilities when generating Advanced Search functionality:

1. The Search region is in the same page as the rest of the Group
2. The Search region is in a separate page.

You control this by setting the **Advanced Search?** property.

There are two properties that will affect the layout of the Search Region:

1. The **formWidth** attribute indicates the width of the Search Region. The default value is 10% which will left align the items. If you set the value to a higher number the items will be located further to the right on the page.
2. The **Advanced Search Layout columns** property indicates in how many columns you want to display your items. By default all the items will be displayed in one column.

Attention: If you use the **formWidth** Property when generating a search region for a page of Form layout, this property values will impact the layout of both the search region and the main form page.

Using query operator

By default, the 'StartsWith' operator is used for String Attributes. In all other cases the equality operator is used.

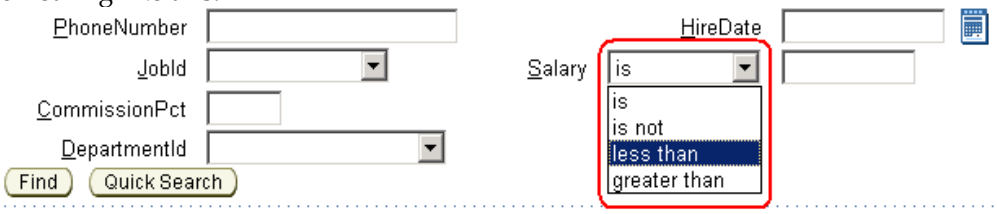
Suggestion: This default behavior is defined in hiddenFormFields.uit

You can change this behavior by setting the **Query Operator** property for an attribute. See the help in the JHeadstart property editor for possible values of this operator.

A special case is the value 'SetByUser' for the **Query Operator. '**SetByUser' means the user of the application can at runtime choose the operator to be used.

Example: Search for a range of values

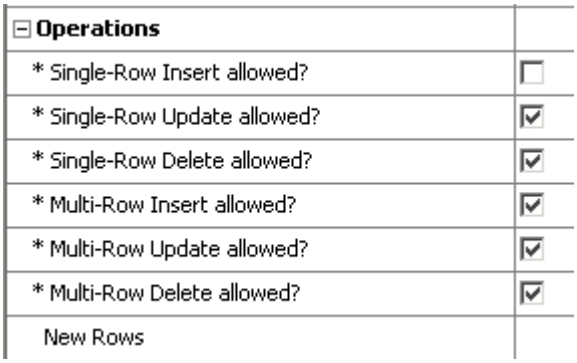
1. Set the **Query Operator** property to 'SetByUser'.
2. Generate the application
3. Go to the 'Advanced Search' region in the generated application. You will see something like this:



4. JHeadstart has generated a poplist with applicable query operators for this field.

Transactional Behavior

This section describes how you can influence the transactional behaviour of generated pages. The properties in the **Operations** group are used for this.



Specifying Insert

Allowing inserting data in a form page

Use property **Single-Row Insert allowed?** JHeadstart will generate a New… button on the page.

Allowing the user to insert data in a table page

When the user is allowed to insert new records in a table one or more empty records should be visible at the bottom of the table. The user can then simply enter the new data in these empty records.

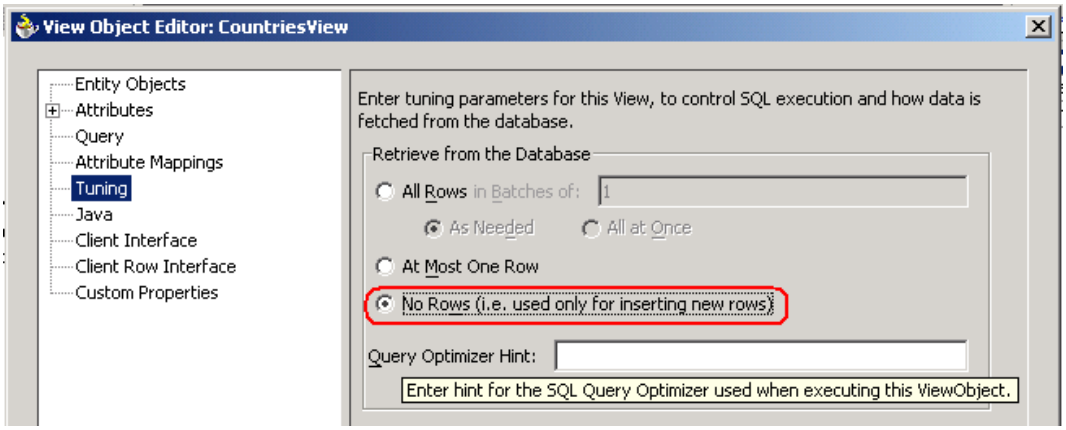
If this is required, then you must use the **New Rows** property in your table group definition and check the **Multi-Row insert allowed?** property.

Build insert only screens

Sometimes you want a page where the user can only enter new records. For example an application for entering new service requests. In this case, a user must not be able to query other data.

In such a case, set **Advanced Search?** and **Quick Search?** properties to none for the group. Disable **Auto Query ?**

When generating, you will get a warning now. To really disable all query functionality for the group, change the View Object query settings. Go to the View Object and change the view into an 'Insert only' View.



Specifying Update

Use properties **Single-Row Update Allowed?** and **Multi-Row Update allowed?** to allow updates in respectively form layouts and table layouts.

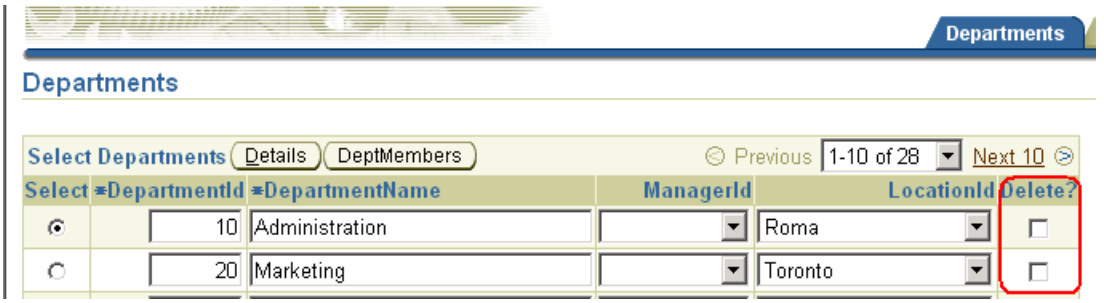
Attention: Multi-row update in a table layout is a feature of JHeadstart. With standard ADF this is not possible.

Specifying Delete

Use properties Single-Row delete allowed? and Multi-Row delete allowed? to allow deletes in respectively form layouts and table layouts.

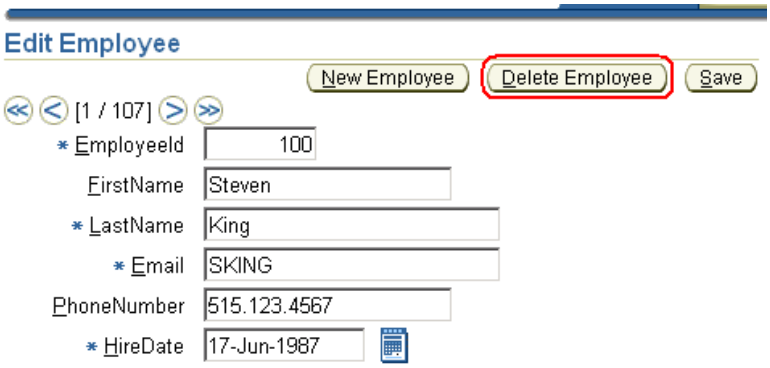
The JHeadstart Application Generator generates a Delete button on a single row page, or a delete check box on a multi row page.

Example 'Delete in a table layout':



The user can now select all the records he wants to delete, and then press the save button to commit the changes to the database.

Example 'Delete in a form layout':

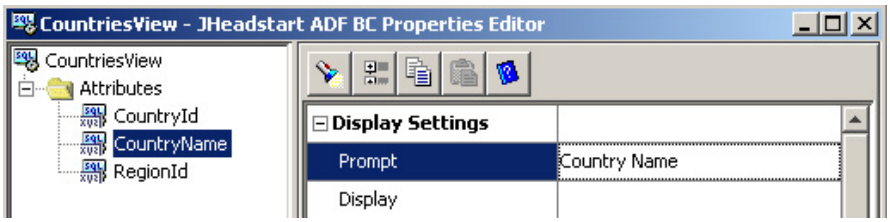


Generating User Interface Widgets

This section describes how you can specify the prompt and default display value of generated items. After that, the various widget types you can generate with JHeadstart are explained.

Specifying the Prompt

By default, the prompt displayed on the screen is the same as the attribute name. If you want to display different prompt, you specify this using the **Prompt** custom property.

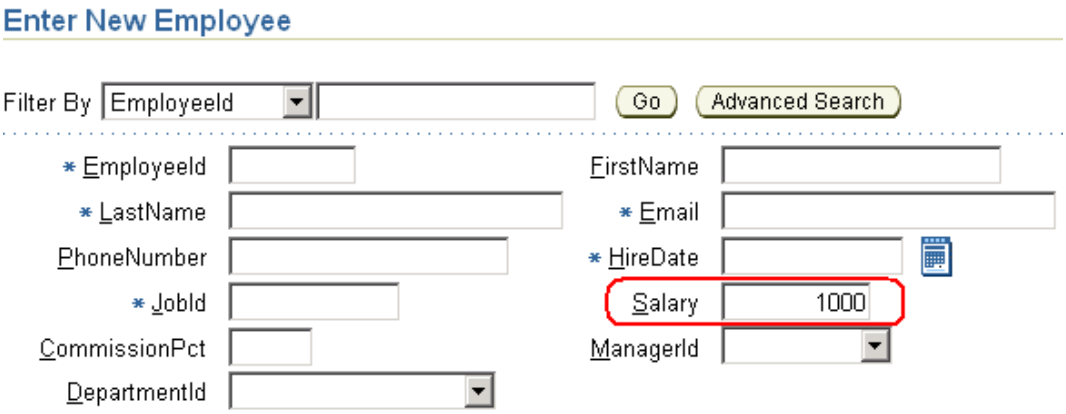


Default Display Value

In the ADF BC Properties Editor you can set the **Default Display Value** of an attribute. This value is used when creating new rows.

For example, a new employee has by default a salary of 1000:

1. Enter the default value with the ADF BC Properties Editor.
2. Generate/run your application and create a new record. The default display value is shown now in the column.



Using EL expressions

In addition to liter values, you can enter Expression Language in the **Default Display Value.**

Imagine that for new employees the salary must be calculated based on job and so on. The result of the calculation (implemented as a DataAction) is stored on the session

context in an object called Salary with method getDefaultSalary. In such a situation enter for Default Display Value the EL `${salary.defaultSalary}`

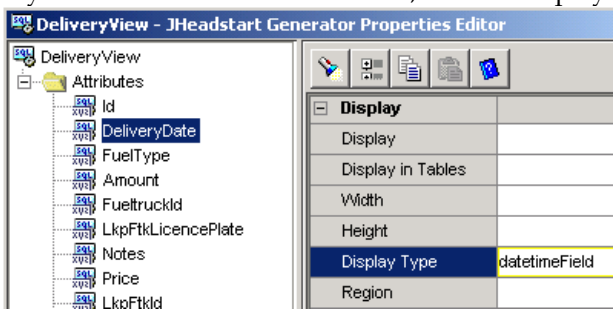
Use the same technique to display the current date: store it on the request or session and enter an EL expression for showing on a page.

Default Display Type

In the ADF BC Properties Editor you can set the **Display Type** of each attribute. However, when you do not set the display type explicitly, JHeadstart will determine a default display type for you. The (somewhat simplified) rules are as follows:

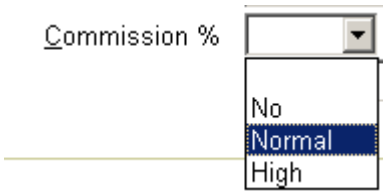
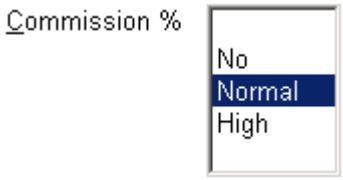
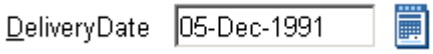
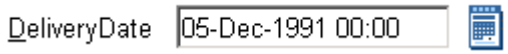
1. If the attribute either has a List Validator, a Domain or a Lookup with displayType not equals to 'lov', the default display type will be 'choice'.
2. Default display type will be 'lov' when the Lookup associated with this attribute has displayType set to 'lov'.
3. When the item is not updateable and the item is neither an image nor an interMedia type, the displayType is set to 'displayField'.
4. If the underlying attribute is of type 'oracle.jbo.domain.Date', the display type is set to 'dateField'.
5. If the underlying attribute is an interMedia type or BlobDomain, then the display type is set to 'fileUpload' if the attribute is updateable, or 'fileDownload' when the attribute is not updateable.
6. If the underlying attribute is of type 'oracle.ord.im.OrdImageDomain', then the display type is set to 'image' when the attribute is not updateable.
7. In all other cases, the display type is set to 'textInput'.

If you want to override the default, use the Display Type property.



You can choose from the following available display types for your attributes:

textInput	* City Roma
checkBox	LPG Enabled ☑

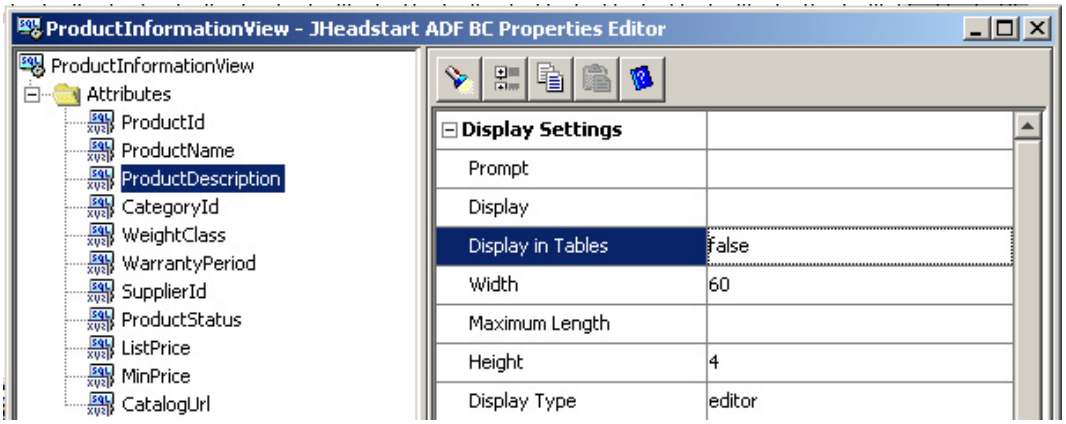
choice	
List	
radio-vertical (or radio-horizontal when using JSP)	
Editor	
dateField	
datetimeField	
displayField	
secret	
fileUpload	
fileDownload	
image	

Generating a Text Item

Define Attribute Display Width and Height

By default, an attribute is displayed with height = 1 (line) and width = the data length of the underlying table column. When the length of the table column is unknown or larger than the value of the service level property **Default Display Width**, the value of this property is used.

To override this you use the Width and Height properties.



Setting Maximum Length

The number of characters that can be entered in the HTML page for an item defaults to the Precision of the underlying attribute. If you want to deviate from this standard you can do this by specifying the **Maximum Length** of the attribute. The value should be the number of characters you require.

Generating a Dropdown List

Use a dropdown list when the list of values the user can choose from is rather small. You have to distinguish between two cases:

- The list of values is static; the values are not queried from the database. In this case you base the dropdown list on a Domain or a List Validator.
- The list of values is dynamic. In this case you must base the dropdown list on a View Object.

Static dropdown list based on a domain

When using this option, you have to add your domain with its values to the domain definition file.

The domain definition file has the following structure:

```
<DomainSet>
  <Domain>
    <AllowableValue>
    </AllowableValue>
  </Domain>
</DomainSet>
```

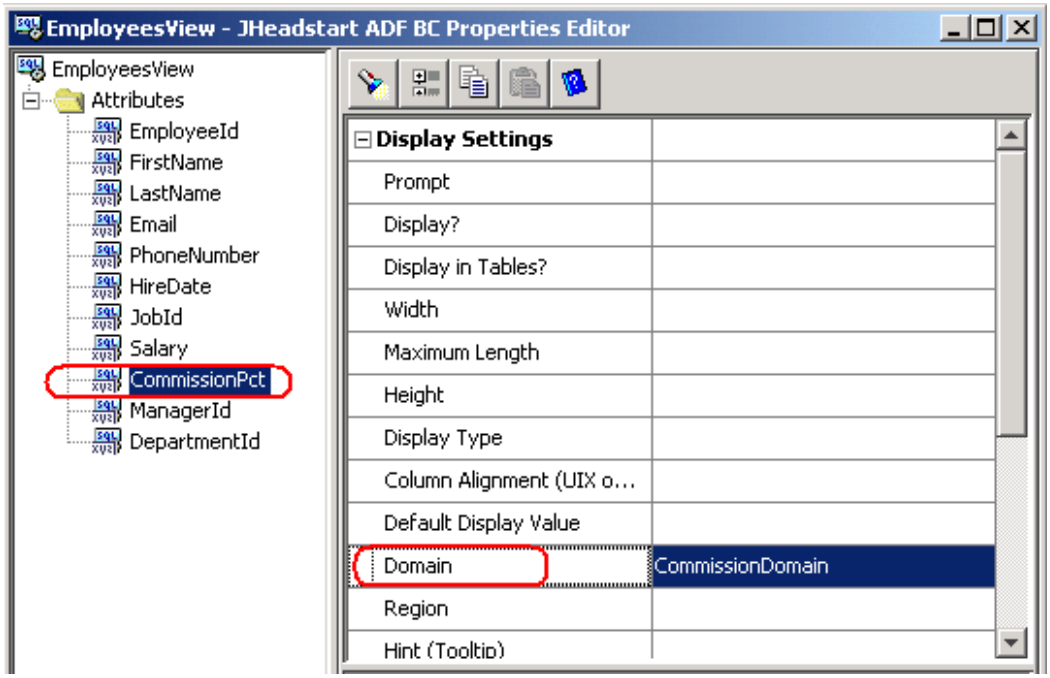
The following syntax rules apply:

- DomainSet has no attributes.
- Domain element has one attribute: name
- Allowable Value has two attributes: value and meaning

Example domain definition file:

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<DomainSet>
  <Domain name="ExampleDomain">
    <AllowableValue value="M" meaning="Male"/>
    <AllowableValue value="F" meaning="Female"/>
  </Domain>
  <Domain name="CommissionDomain">
    <AllowableValue value="0.0" meaning="No Commission"/>
    <AllowableValue value="0.1" meaning="Low"/>
    <AllowableValue value="0.2" meaning="High"/>
  </Domain>
</DomainSet>
```

When the domain is available in the Domain Definition File, then you must ensure that the appropriate attributes will use the domain by setting the **Domain** property.



Generate your application, and you will get a dropdown list by default. There is no need to set the **Display Type** property.

Translation of static domains

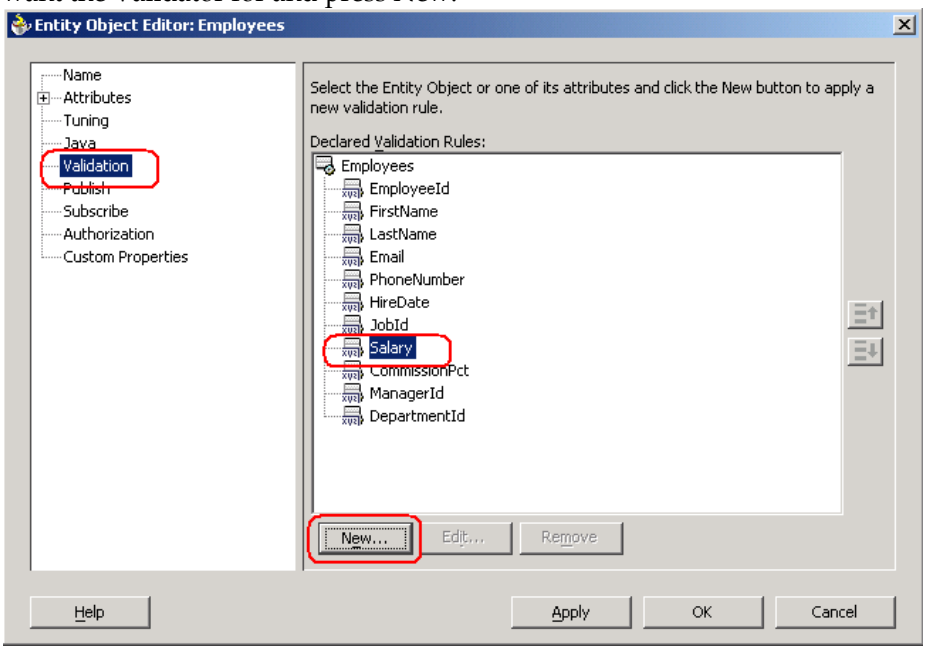
As you see, the meaning of the domains in the domain definition file is only in 1 language. When you need to be able to translate domain meanings in other languages, set service level property **Generate NLS-enabled prompts and tabs** to true. When this property is set, JHeadstart will generate entries for each domain value in the ApplicationResources.properties file.

Static dropdown list based on a List Validator

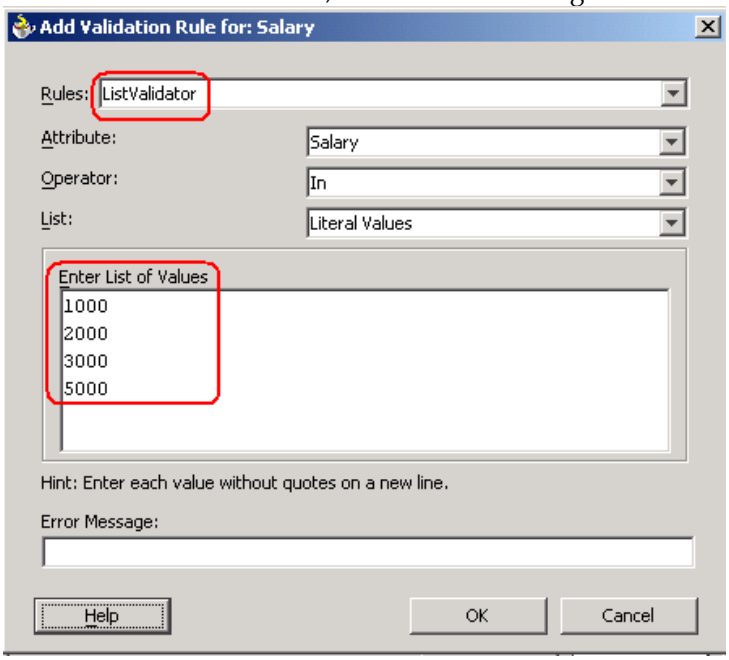
ADF BC has the possibility to add Validators to Entity Objects. With these Validators, you can check for allowable values in the Model layer.

JHeadstart can use List Validators to generate dropdown lists. See the steps below. In this example we use a List Validator to check the allowable values for the salary column.

1. Edit the Entity Object and go to the Validation Node. Select the attribute you want the Validator for and press New:



2. Choose List Validator and enter the Allowable Values. Note that you can only enter allowable values here, and not the meaning.



3. At the service level, set the property **Generate NLS-enabled prompts and tabs** to true. By doing so, JHeadstart will add entries to the Resource Bundle.

4. Generate the application (you will get a dropdown list by default). Lookup the added entries in the generated Resource Bundle and enter the correct meaning. In this example, the entries are called DOMAIN_EMPLOYEES_SALARY_<value>

Attention: You may ask when to use the domain definitions file, and when to use a List Validator. Differences are:

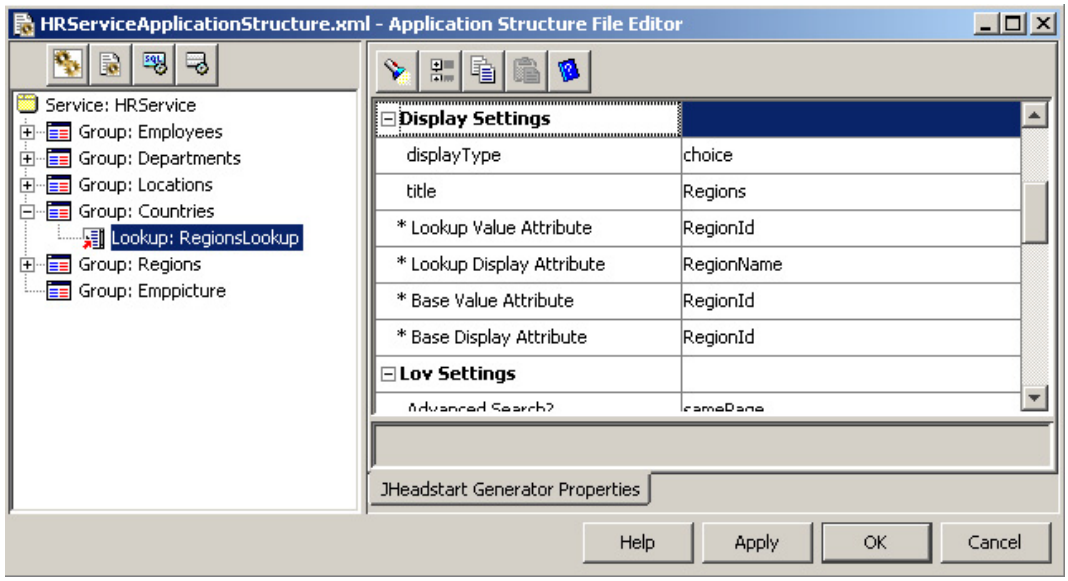
- You cannot reuse a List Validator. You have to define a new one for each attribute you want to validate, whereas you can use one domain throughout your application.
- A List Validator also enforces the check in the Model layer.
- In the domain definition file, you can define a meaning for each allowable value.

Dynamic dropdown list based on a lookup

When the list of values must be dynamic, use a lookup based on a View Object to generate the dropdown list.

Steps to generate a dropdown list based on a View Object:

1. Create a Lookup based on the View Object. Select the View Object you want, by setting the **View Object Usage** property for the Lookup.
2. Set **displayType** to 'choice'.
3. Select the attribute you want to see in the dropdown list, by setting **the Lookup Display Attribute** property.
4. Set Lookup Value Attribute to the name of the lookup attribute that should be stored as value in the Base Value Attribute.
5. Set both **Base Value Attribute** and **Base Display Attribute** to the name of the base group attribute that should get the value of the Lookup Value Attribute.



Generating a Radio Group

Use a radio group when the list of values the user can choose from is small. You have to distinguish between two cases:

- The list of values is static; the values are not queried from the database. In this case you base the radio group on a Domain or a List Validator.
- The list of values is dynamic. In this case you must base the radio group on a View Object.

Static radio group based on a domain

When using this option, you have to add your domain with its values to the domain definition file.

The domain definition file has the following structure:

```
<DomainSet>
  <Domain>
    <AllowableValue>
    </AllowableValue>
  </Domain>
</DomainSet>
```

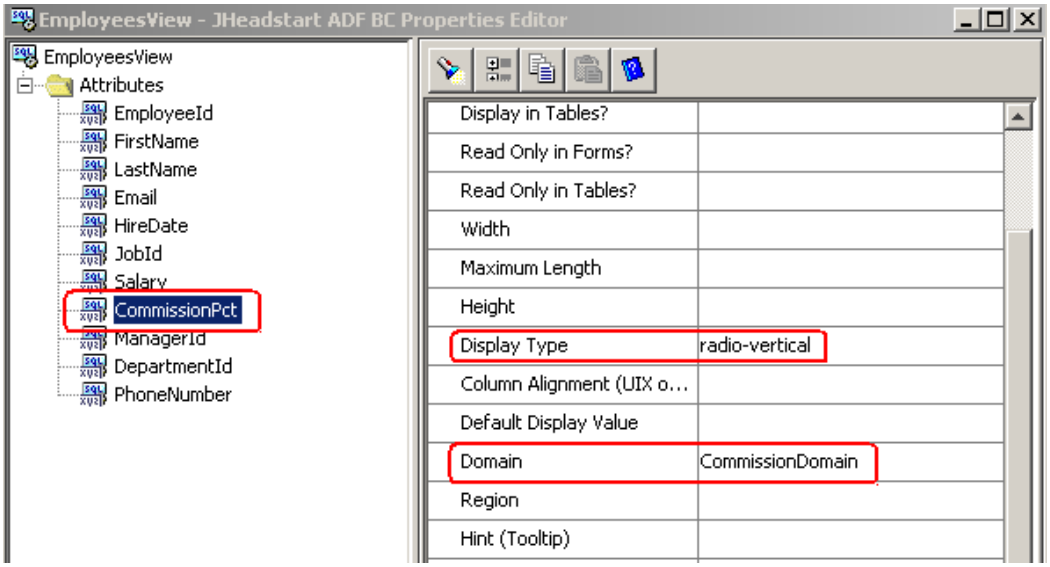
The following syntax rules apply:

- DomainSet has no attributes.
- Domain element has one attribute: name
- Allowable Value has two attributes: value and meaning

Example domain definition file:

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<DomainSet>
  <Domain name="ExampleDomain">
    <AllowableValue value="M" meaning="Male"/>
    <AllowableValue value="F" meaning="Female"/>
  </Domain>
  <Domain name="CommissionDomain">
    <AllowableValue value="0.0" meaning="No Commission"/>
    <AllowableValue value="0.1" meaning="Low"/>
    <AllowableValue value="0.2" meaning="High"/>
  </Domain>
</DomainSet>
```

When the domain is available in the Domain Definition File, then you must ensure that the appropriate attributes will use the domain by setting the **Domain** property. The Display Type property must be set to radio-vertical or radio-horizontal. Note that radio-horizontal only works with JSP, in UIX pages you will always get a vertically oriented radio group.



Generate your application, and you will get a radio group.

Translation of static domains

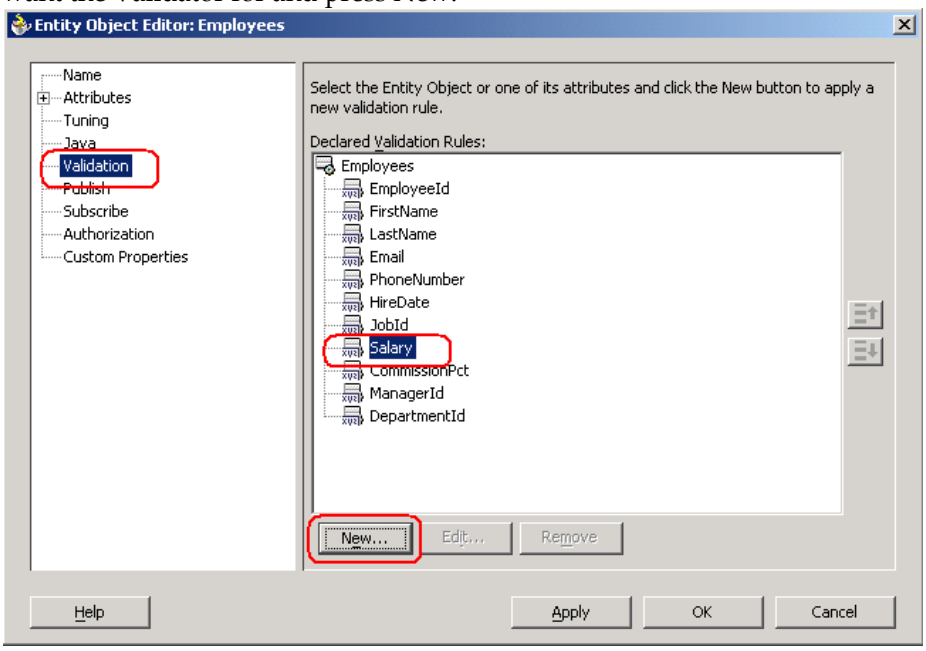
As you see, the meaning of the domains in the domain definition file is only in 1 language. When you need to be able to translate domain meanings in other languages, set service level property **Generate NLS-enabled prompts and tabs** to true. When this property is set, JHeadstart will generate entries for each domain value in the ApplicationResources.properties file.

Static radio group based on a List Validator

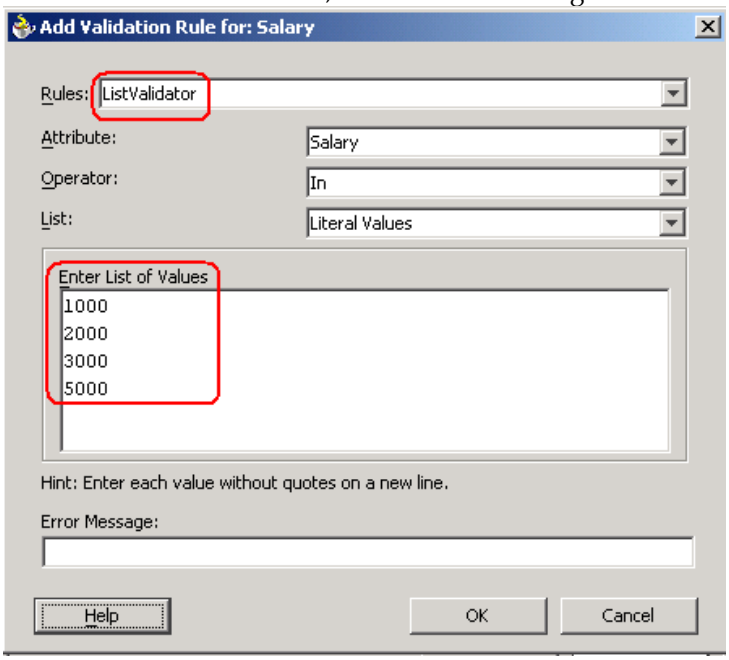
ADF BC has the possibility to add Validators to Entity Objects. With these Validators, you can check for allowable values in the Model layer.

JHeadstart can use List Validators to generate dropdown lists. See the steps below. In this example we use a List Validator to check the allowable values for the salary column.

1. Edit the Entity Object and go to the Validation Node. Select the attribute you want the Validator for and press New:



2. Choose List Validator and enter the Allowable Values. Note that you can only enter allowable values here, and not the meaning.



3. Set the **Display Type** property of the attribute to "radio-vertical" or "radio-horizontal".

4. At the service level, set the property **Generate NLS-enabled prompts and tabs** to true. By doing so, JHeadstart will add entries to the Resource Bundle.

5. Generate the application. Lookup the added entries in the generated Resource Bundle and enter the correct meaning. In this example, the entries are called DOMAIN_EMPLOYEES_SALARY_<value>

Attention: You may ask when to use the domain definitions file, and when to use a List Validator. Differences are:

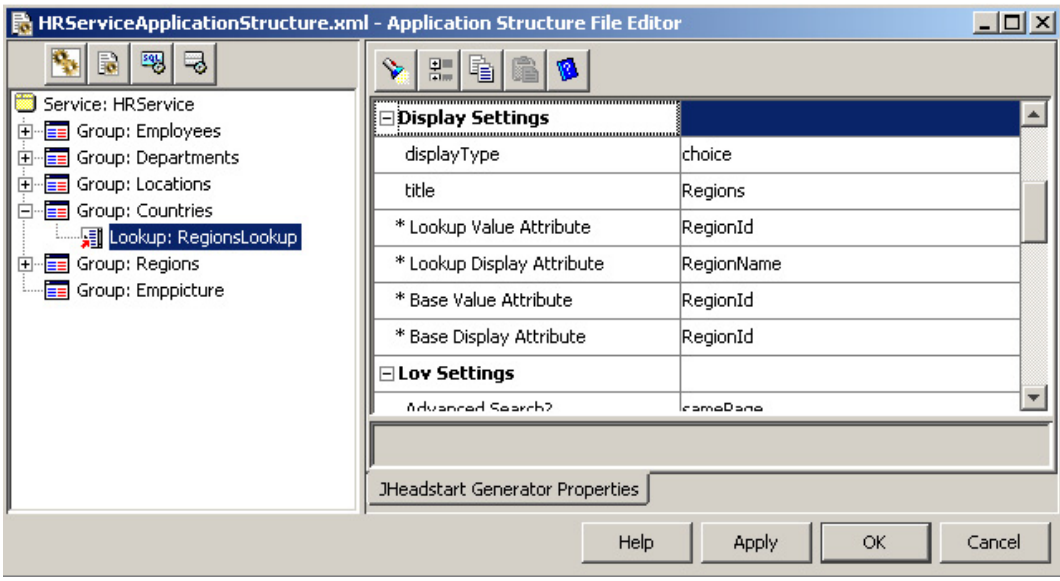
- You cannot reuse a List Validator. You have to define a new one for each attribute you want to validate, whereas you can use one domain throughout your application.
- A List Validator also enforces the check in the Model layer.
- In the domain definition file, you can define a meaning for each allowable value.

Dynamic radio group based on a lookup

When the radio group must be dynamic, use a lookup based on a View Object to generate the dropdown list.

Steps to generate a radio group based on a View Object:

1. Create a Lookup based on the View Object. Select the View Object you want, by setting the **View Object Usage** property for the Lookup.
2. Set Lookup Value Attribute to the name of the lookup attribute that should be stored as value in the Base Value Attribute.
3. Set both **Base Value Attribute** and **Base Display Attribute** to the name of the base group attribute that should get the value of the Lookup Value Attribute.



4. Set the **Display Type** property of the Base Value Attribute to "radio-vertical" or "radio-horizontal".

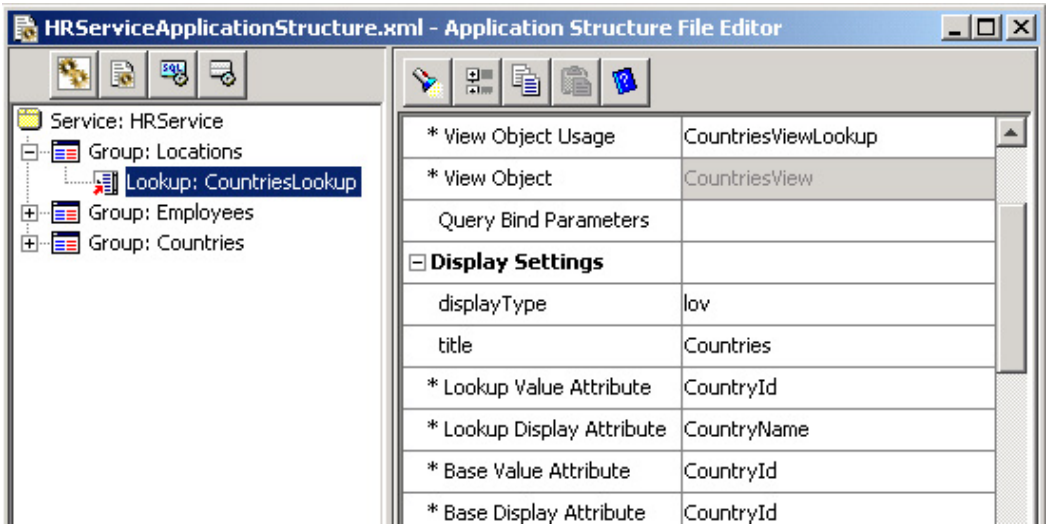
Note that the **Display Type** property of the Lookup itself is ignored when the base value attribute has a display type of "radio-vertical" or "radio-horizontal". In a future release, we will add radio-vertical and radio-horizontal as allowable values for the Lookup Display Type.

Generating List of Values

Use a list of values (lov) when you have a lookup to a related table and the number of records in the related table is too big for a drop down list or you want to provide search functionality on the lookup.

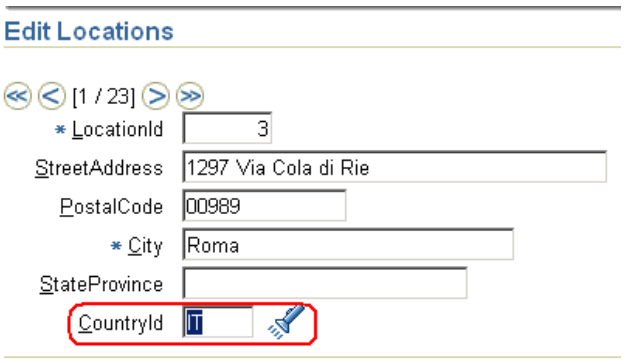
Steps to create a list of values:

1. Make sure you have a lookup with the correct **View Object Usage**.
2. Set **Display Type** to 'lov' for the lookup.
3. Choose the attribute you want to see in the lov window by setting the **Lookup Display Attribute** property.



Add attributes from other Entities to a View object

There are situations where you will need to define lookup attributes in the base view object. Take a look at the screen below. The CountryId column is part of the Locations View Object. The CountryId is a foreign key referencing the Countries View Object. However, in many (most) cases, you do not want to show the foreign key column, particular in the case of artificial keys. Instead you want to show a more meaningful field from the referenced table, in this case the Country Name.

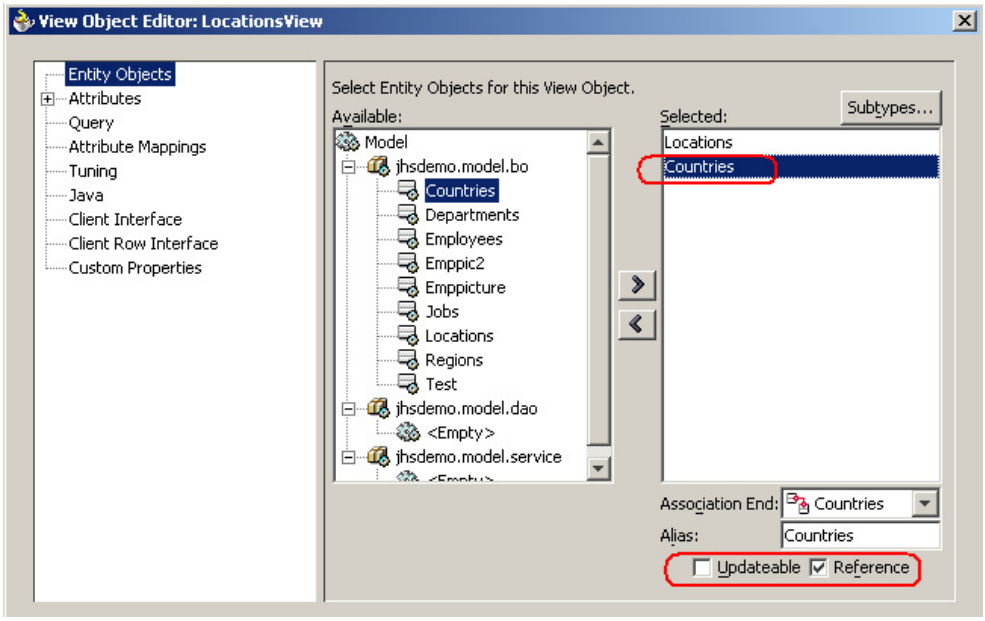


In that case you must include all the attributes you want to display on the page to become a part of the view object on which you base your group in the Application

Structure File. So the CountryName attribute of the Countries View must be added to the LocationsView.

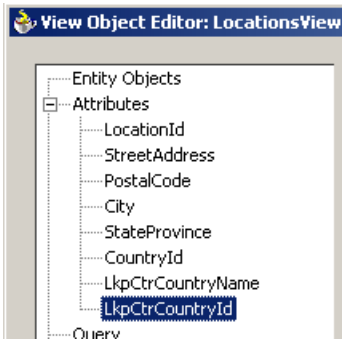
Perform the following steps to accomplish this:

1. Select the view object you want to modify, right mouse click, and select Edit <view Object>
2. Navigate to the Entity Objects node. You will notice that your base entity is at the right hand side as the selected entity. To be able to include lookup attributes you must select the lookup entity and move it from the Available list to the Selected list.



3. Set a proper alias for the lookup entity and select the right association end.
4. Navigate to the Attributes node. You will notice that all the attributes of the base entity take part of the selected list. Now, in the Available list, select those attributes from the lookup entity object you want to display in the view object, and move them to the Selected list.
5. The key attribute from the lookup entity (let's call it the Lookup Key) is always included and usually ends up with a strange name. If for example it is called 'Id', it will be named 'Id1' on the base table. This is not a good name. However, the name XxxId (where Xxx is the entity alias) is already used by the Foreign Key attribute of the base entity. On the 'Attribute Settings' node, rename the Id1 attribute using the naming convention LkpXxxId to avoid confusion.
6. It is also a good practice to rename the other lookup attributes so they are prefixed with the entity alias. This makes them easily identifiable as lookup

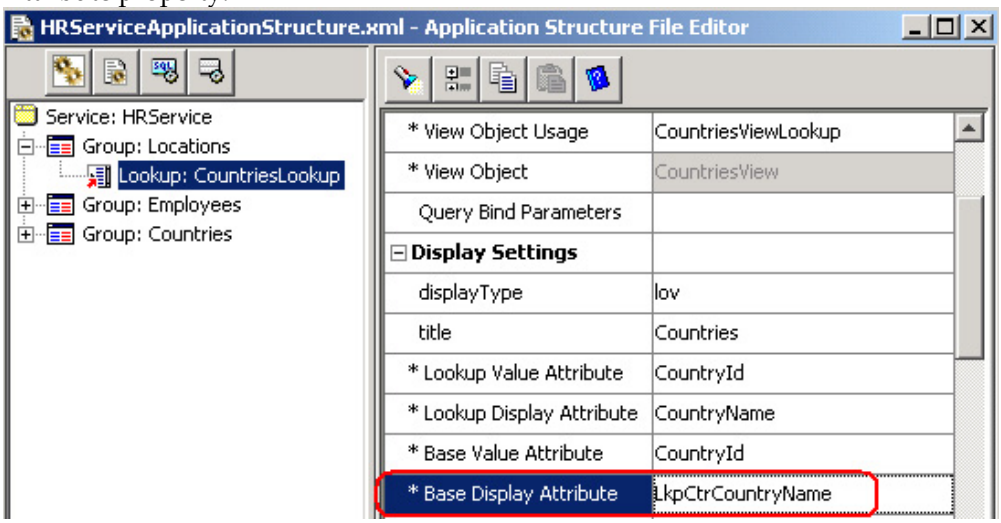
attributes.



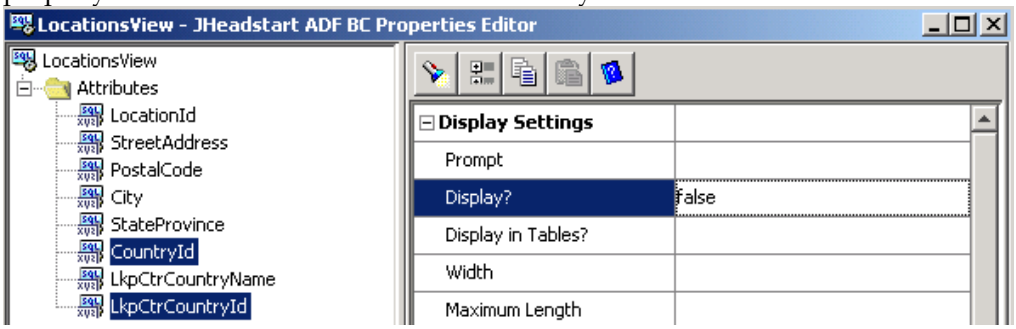
7. Test the View Object with the ADF BC Tester. Check whether the LkpCtrCountryName attribute changes when you change the CountryId attribute.

Now we have extended the View Object, we need to make some changes to our page definition:

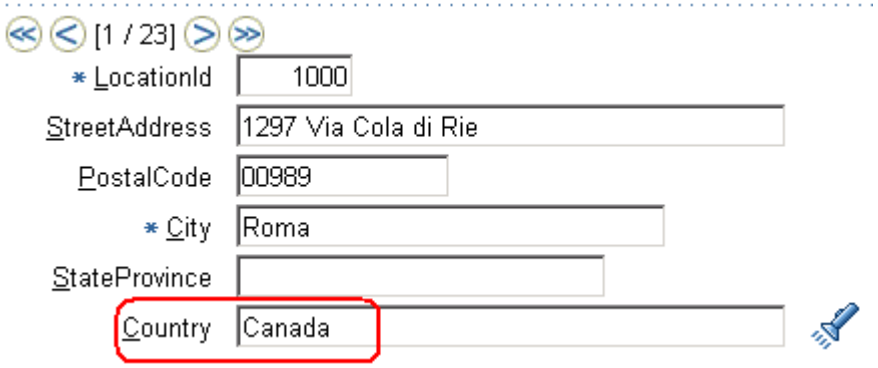
1. Select the attribute you want to show on the base page by setting de Base Display Attribute property:



2. Hide the columns you do not want to show anymore by setting the **Display** property to false. In this case we hide the CountryId attribute.



3. Generate and you will get this:



Attention:

1. When your View Type is JSP, the fields from the lookup table are read-only. So the user must use the LOV to select a value.

2. When your View Type is UIX, set Use **LOV for validation?** To true. See Use LOV for validation

Displaying multiple attributes in List of Values

When you click on the **Lookup Display Attribute** property , you get a drop down list to select an attribute. This attribute will be displayed in the List of Values window. If you want to display multiple attributes in the List of Values window, you can do so by typing in additional attribute names, separated by a comma, in this property:

Display Settings	
displayType	lov
title	Employees
* Lookup Value Attribute	EmployeeId
* Lookup Display Attribute	FirstName, LastName, HireDate
* Base Value Attribute	ManagerId
* Base Display Attribute	ManagerId

When you generate the application with the above settings, the List of Values will look like the screen shot below.

Use LOV for Validation

Note: This section is only applicable when your **View Type** is set to UIX.

A List of Values is normally used to assist the user in selecting a value for a foreign key column. The user can navigate to the List of Values, type some search criteria and select the value from the list and navigate back to the main page.

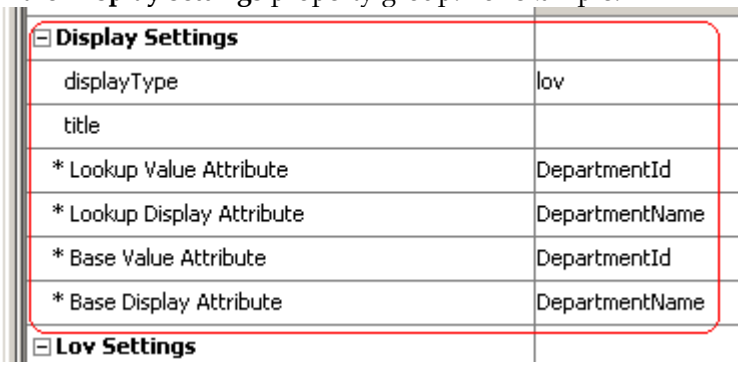
However, in most cases, the user will know many values by heart, and needs the List of Values only for special cases, for example values that are infrequently used. With the Use LOV for Validation functionality, JHeadstart can generate pages that assist the user in both cases. It works this way:

1. The user enters (part of) the lookup attribute value.
2. The JHeadstart runtime checks how many records in the lookup match the value the user entered.
3. When it is exactly one, the list of values window is not shown, but the JHeadstart runtime finds the matching record and auto-completes the entered value.
4. When zero or more than 1 records in the lookup match the entered value, automatically the list of values window is launched and pre-queried with the value the user entered.

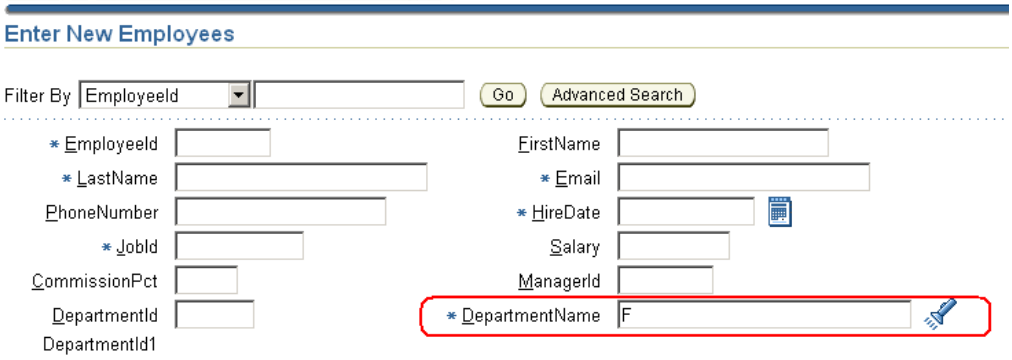
So, the system decides whether the list of values should be launched. This saves the user from manually invoking the list of values and thus improves end-user productivity.

The next steps instruct you how to build this. The EMPLOYEES and DEPARTMENTS tables are used as example.

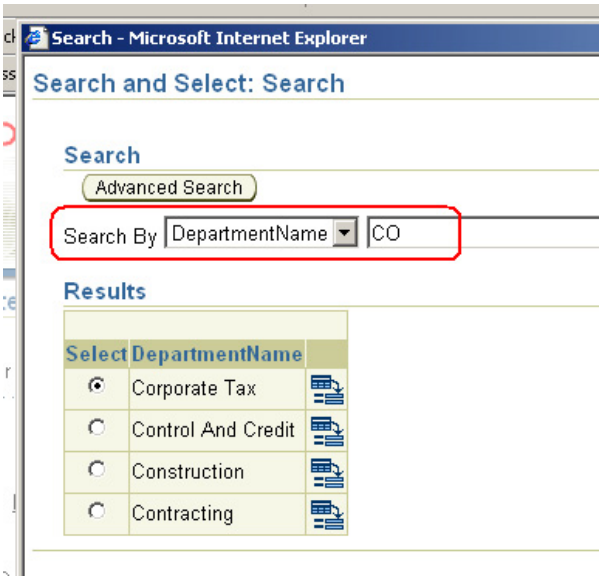
1. Extend the base View Object you want to manipulate with the descriptor attributes of the lookup View Object. See Defining the Lookup Attributes in the view object for instructions. In our example, the DepartmentName attribute should be added to the EmployeesView View Object.
2. In the JHeadstart Application Structure File, define a group for the base View Object and a lookup for the lookup View Object.
3. Set group properties as you like.
4. In the lookup, set **displayType** to lov. Set correct values for the other attributes in the **Display settings** property group. For example:



5. In the lookup, set **Use LOV for Validation? (UIX only)** to true.
6. Generate the application. You might get this:



- Navigate to the DepartmentName, enter 'F' in the field and press TAB. Because there is only one department name starting with F, no LOV will be launched and the department name is auto completed.
- Navigate to the DepartmentName, enter 'CO' in the field and press TAB. Because multiple department names start with CO, the list of values is launched and prequeried.



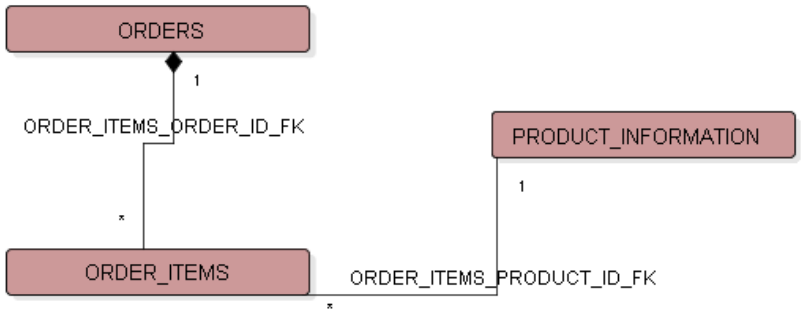
Warning: There is one situation where using the LOV for validation does not work: in new rows of a table layout. In those rows you have to click the LOV icon to use the List of Values. If you try LOV for validation in such a new table row, you will get a JavaScript alert explaining this.
Note that this limitation does not apply to new rows in form layout. There the LOV can be used for validation without any problems.

Selecting multiple values in a List of Values

Note: This section is only applicable when your **View Type** is set to UIX.

JHeadstart can generate a List of Values where the user can select many values at once. This improves the usability of the application.

Suppose you have this data model:



An ORDER has multiple ORDER_ITEMS, so you can order multiple PRODUCTS in one order. Without multi-select, the user has to create a new ORDER_ITEMS records and select the PRODUCT for that ORDER_ITEM. Imagine the time needed to enter an ORDER with say 20 products.

With multi-select List of Values, the user selects all the products for the ORDER at once. When returning in the main page, multiple new rows are created AT ONCE. Of course, this is only possible when ORDER_ITEMS is a table layout.

How to generate this:

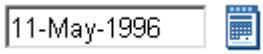
1. Because multiple ORDER_ITEMS are created at once, you MUST have added to your Business Components Model the ability to automatically generate the primary key values. In this case, the Business Components layer should generate the LINE_ITEM_ID of the ORDER_ITEMS. See Prepare Model for generation for instructions.
2. Make sure you have group definitions right. In this example, ORDERS can have form layout, ORDER ITEMS MUST have table layout and PRODUCT_INFORMATION is a lookup.
3. In the base group of the lov, give the **New Rows** property a value greater than zero.
4. In the lookup, set **displayType** to lov. **Set Allow multiple selection in lov? (UIX only)** to true.
5. Generate the application.

Attention: If you combine a Multi-Select LOV with Use Table Ranges, then it can occur that some of the newly created rows are not immediately visible, they have moved to the next table range.

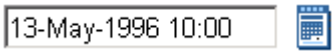
For example, suppose you have a multi-select LOV in a table page with table range size = 10. Suppose that you are showing rows 11-20 of 50, and 2 empty rows for creating new records. If you now use the multi-select LOV in one of the empty rows to create 3 new rows, the first new row will be visible at bottom of current table range (position 10). The second and third new row will be in the next table range, at positions 1 and 2. The row that was originally at position 10 of the current table range, has now been moved up to position 3 of the next table range. The current table range will show rows 11-20 of 53.

Generating a Date (time) Field

By default, you will get display type 'dateField' when the attribute in the ViewObject is of type 'Date'. In a dateField you can enter only the date.



You can change the **Display Type** property for an attribute to 'dateTimeField'. In a dateTimeField you can enter a date and a time.



Specifying display format for date and datetime field

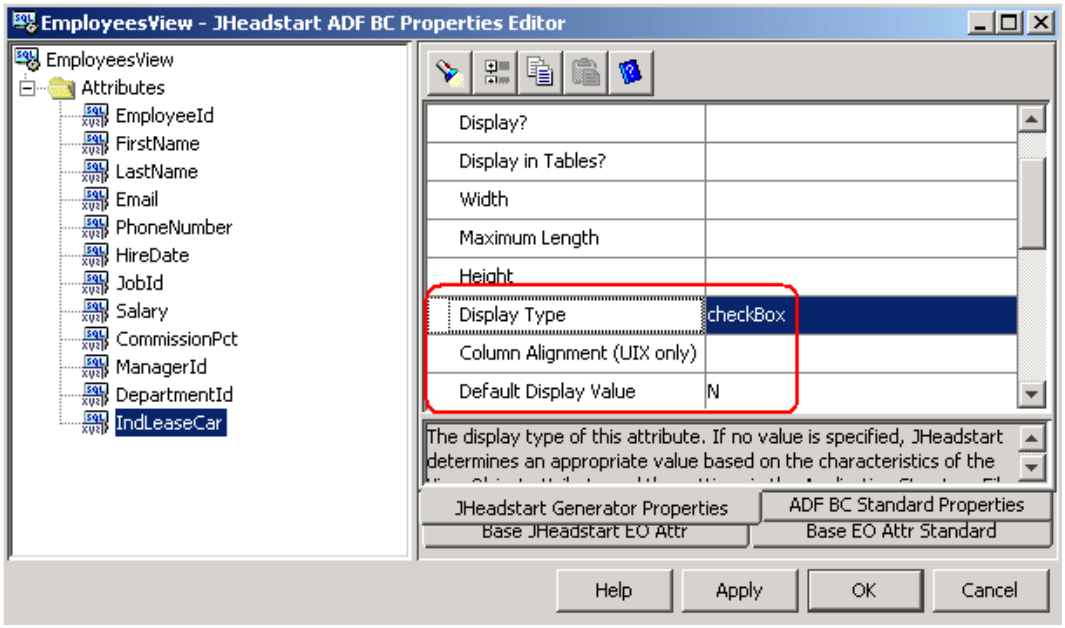
By changing the service level properties **Date Format** and **DateTime Format**, you can define the display format of both dates and datetimes. The format strings used here, are as defined in `java.text.SimpleDateFormat`. The JAG takes the values of these properties and puts them in a dateformat.properties file. This file is used at runtime.

Generating a Checkbox

You can generate a checkbox for attributes that have exactly two allowable values: one value is shown as checked, and the other as unchecked. Because the HR sample schema does not have such an attribute, we have added IND_LEASE_CAR to the EMPLOYEES table, with allowable values Y and N.

Steps to generate a checkbox:

1. Add a domain to the domain definitions file with the two values:
```
<Domain name="BooleanDomain">
  <AllowableValue value="Y" meaning="Yes"/>
  <AllowableValue value="N" meaning="No"/>
</Domain>
```
2. Set **Display Type** to 'checkbox'.
3. Set **Domain** to 'BooleanDomain'.
4. Set **Default Display Value** to the attribute value you want to show as an unchecked checkbox.



Attention: In a search region, IndLeaseCar will show as a dropdown list and not as a checkbox. The reason is that we have three situations when searching:

1. We want to search for records with IndLeaseCar='Y'
2. We want to search for records with IndLeaseCar='N'.
3. We do not want to consider the value of IndLeaseCar in the search, but are searching on other criteria.

File Upload, File Download and Showing Image Files

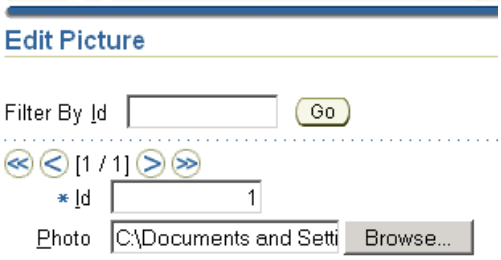
You can generate File Upload, File Download and/or Showing Image Files for columns of types BLOB, ORDSYS.ORDIMAGE and ORDSYS.ORDDOC

Suggestion: ORDSYS.ORDImage is an object type defined in the Oracle interMedia feature of the Oracle database. ORDImage can process and automatically extract properties of images of a variety of popular data formats.
The ORDSYS.ORDDoc type can store any heterogeneous media data including audio, image, and video data in a database column.
See the interMedia documentation of the Oracle RDBMS for more information.

By default, JAG will generate a **file upload** field for these column types.

Example of generating a **file upload** field in the HR sample schema for uploading photo's of employees:

1. Make sure you have a table with a column of the correct datatype. This is sufficient:
   create table emppicture (id number, photo ordsys.ordimage);
   alter table emppicture add constraint pic_pk primary key (id);
2. Add an ADF Entity Object and ADF View Object for this table. Add the View Object to the ADF Application Module. Add a group to your JHeadstart Application Structure file for this new View Object and generate your application. You will get something like this:

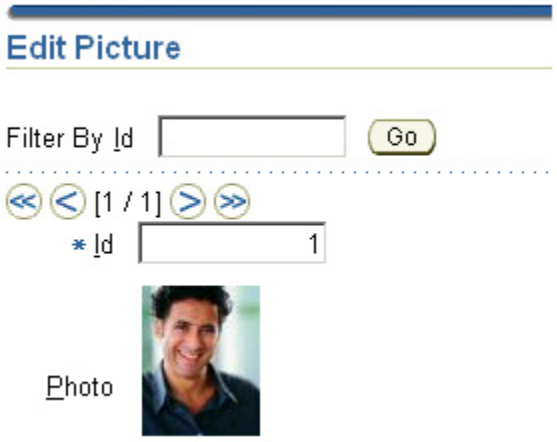


With the **Browse** button you can select the image file you want to upload for this record.

If instead, you want that field to be a **file download**, you can change the Display Type of the ADF Attribute to fileDownload. If the file stored in the database column is an image that can be shown in HTML (like for example *.gif or *.jpg files), you can set the Display Type to **image**.

Attention: Set correct values for **Width** and **Heigth** properties when **display Type** is image.

After changing the **Display Type** to image and regenerate, you will get this:



If you want to limit the size of the file that can be uploaded, you can do this as a post-generation step, by adding the "maxFileSize" property to the <controller> element in the struts-config:

```
<controller
processorClass="oracle.jheadstart.controller.struts.JhsRequestProcessor"
 forwardPattern="$P" maxFileSize="500K" locale="true"/>
```

The maxFileSize can be expressed as a number followed by a "K", "M", or "G", which are interpreted to mean kilobytes, megabytes, or gigabytes, respectively. The default value is 250M.

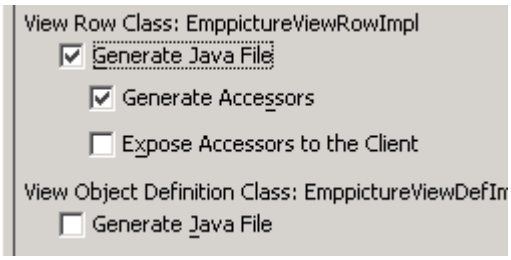
Combining File Display Options

Normally, you can only choose one of the three options (file upload, file download or image) for a particular BLOB/OrdImage/OrdDoc attribute. If you want to implement two or more of these file display types for one attribute, you must create one or more Transient Attributes in the ADF View Object.

Attention: If the display types to be used include fileUpload, put that display type always on the bound attribute (that is, the attribute that is linked to an Entity attribute). Of course, it makes no sense to upload images in a transient attribute.

For generating Upload, Download and Show Image at once, create Transient Attributes as follows:

1. Edit the ADF View Object
2. Go to the Attributes page
3. Click the button New…
4. Specify a name for the new attribute (for example: DownloadPhoto) . Choose the attribute type (Blobdomain or OrdImageDomain or OrdDocDomain) and click OK.
5. Repeat steps 3 and 4 for a transient attribute called ShowPhoto.
6. Go to the Java page.
7. Ensure that Generate Java File is checked for the View **Row** Class, and that Generate Accessors is checked.



8. Press Finish
9. Open the generated ViewRowImpl.java file (open the View Object node to see the file) and go to the get<newAttributeName> methods.

   Change the generated code of the get methods of the add transient attributes so they do not return the transient attribute but the bound attribute.
   Generated code:

```
public OrdImageDomain getDownloadPhoto()
{
  return (OrdImageDomain)getAttributeInternal(DOWNLOADPHOTO);
}
```

   Change to:

```
public OrdImageDomain getDownloadPhoto()
{
  return getPhoto();
}
```

10. We have now three transient attributes, all returning the value of the same database column. So we can set different display types on each of the attributes. Set the display type of the transient attributes to fileDownload and image respectively.
11. Regenerate.

Store additional info of uploaded files

When uploading files, you can also save the name, size and mimetypes of the uploaded files in the database. Add attributes for name, size and mimetype to your View Objects. Set the properties **File Name Attribute**, **File Size Attribute** and **File Mime Type attribute** for the upload attribute.

Attention: When you specify the **File Name Attribute** property on the upload attribute, you must also set this property on the corresponding transient attribute that you might have created to generate a download link. If you forget this, the download link will **not** display the actual file name, but the attribute prompt instead which is the same for all rows.

Note that the interMedia objecttypes have built-in functions to retrieve similar information. For example, getmimetype() and getcontentlength() return the mimetype and size of a stored interMedia object. So there is no need to have additional columns to store mimetype when using interMedia. You can add these functions to the View Object as a calculated attribute

Customizing Page Layout Generation

This section describes how you can even further customize the generation of applications. For example, when you want to have your own look and feel generated.

Using Generator Templates

Introduction

When generating the application, JAG uses templates as a basis. So, the layout and the behavior of the various page types are defined in templates. When generating, JAG replaces placeholders in the template with content taken from the Application Structure File.

The JHeadstart templates are stored in the template directory of your JHeadstart project. There are two groups of files:

- *.jjt: The templates used for generating JSP.
- *.jut: The templates used for generating UIX.

The JhsTemplates.jtp file describes all the available templates and has pointers to the location of the templates. So, when changing the shipped JHeadstart templates, you can copy them to another location and change the filenames in the JhsTemplates.jtp file.

On the service level, you can define the template to be used for the whole service, by setting to **Template Properties File** property. Use this setting when defining the template for the whole service.

On individual groups, you can override the service setting for that particular group. Use the group level **Template Properties File** property in that case. Use this setting when you need different page behavior for one particular page.

Attention: Make sure not to confuse the JHeadstart templates used as a basis for generation with the templates used in the UIX View technology:

1. JHeadstart templates have extension .jjt or .jut and are stored in the template directory. They are only used during generation of the application, not at runtime.

2. UIX templates have extension .uit and are stored under the WEB-INF directory. They are used at runtime when UIX generates pages from UIX files.

Example

In this example, we will make an extremely simple Look and Feel customizing, only to show how to use templates.

1. Examine a generated page with form layout. Notice the order of the buttons. The New record button is placed before the Save Button. We will reverse the order.

2. Copy JhsTemplates.jtp to JhsTemplatesCustom.jtp. Change the **Template Properties File** property for the group to JhsTemplatesCustom.jtp
3. Open JhsTemplatesCustom.jtp. Change the line
`dataPage=template/dataPage to`
`dataPage=template/dataPageCustom`
4. Copy template dataPage.jut to dataPageCustom.jut (use jjt version when generating for JSP).
5. Open dataPageCustom.jut in JDeveloper. Locate the NEW_BUTTON placeholder. Change the file, so that NEW_BUTTON is on the line below $SAVE_BUTTON$
6. Regenerate the application. The order of the buttons should have changed now.

Use of properties in templates

Any property in the Application Structure File can be used in the templates. Imagine you want to show the Display Title (Plural) of the group somewhere in a page. In the template you refer to the value of this property with placeholder $GROUP_DISPLAY_TITLE$. At generation, this placeholder is replaced with the value of the property in the Application Structure File. You can use the following placeholders:

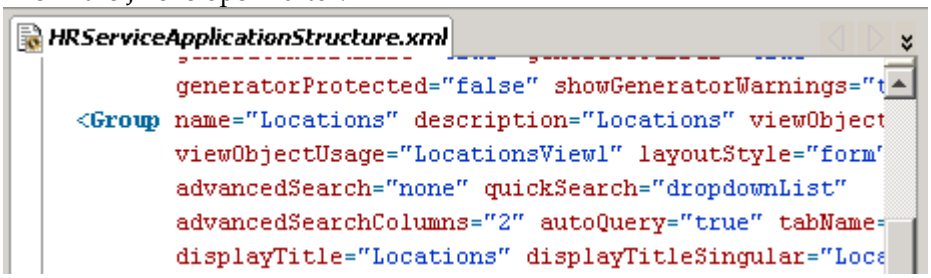
1. $GROUP_<name of a group level property>$ to refer to a group level property
2. $SERVICE_<name of a service level property> to refer to a service level property
3. $PARENT_<name of a group level property>$ to refer to a property of the parent group.
4. $LOOKUP_<name of a lookup property>$ to refer to a propery of a lookup.
5. $CHILD_GROUP_<childgroupnumber>__<name of a group level property>$ to refer to a property of a child group. Child group numbers start with 1.
6. By prefixing the placeholder name with $NLS, JHeadstart generated entries in the Resource Bundle, so you can translate.

Furthermore, two special tokens exist:

1. $GROUP_NR$ is the sequence number of the group in the Application Structure File.

2. $CHILD_GROUP_<child group number>_EXISTS$ returns true or false depending on the existence of a child group with the specified number.

You can find the name of the property you need by opening the Application Structure File in the JDeveloper Editor:



You have to change the XML attribute names somewhat to get the placeholder name. For example, viewObjectUsage becomes VIEW_OBJECT_USAGE.

Overriding token generation

The generator templates also contain tokens that drive the generation proces. For example, the token $FORM_ITEMS$ is used to indicate the place where a form must be generated. You can override the default behavior of tokens as follows:

1. Add the token you want to override to JhsTemplates.jtp or the group specific template. The value of the token is the location you of the code snippet you want to be generated. For example: `FORM_ITEMS=template/mycustomform.uix`
2. Define the content of template/mypostgenchange.uix.

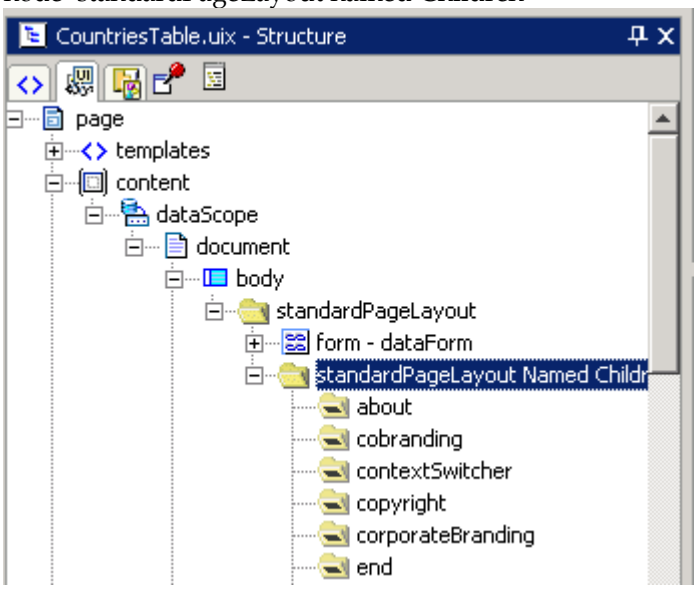
This feature is very useful for preserving post generation changes. After the initial generation, you have made some changes to get exactly the layout you want. To preserve those changes, save only the part of the page you have changed in a file (called template/mypostgenchange.uix for example) . Define a custom template for that group and override the FORM_ITEMS behavior as described above. When generating again, your post generation change is preserved.

Custom properties and templates

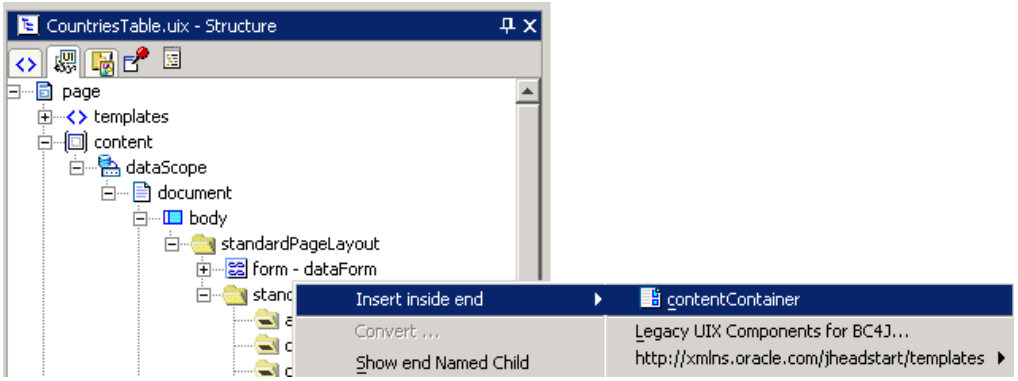
In this section we will add a help region to generated pages. Also, we will see how we can use custom properties to extra info within generated pages. We will first do the change by hand by doing a post generation change in one of our generated pages. When we are satisfied with the results, we will capture our handmade change and implement the same change in the JHeadstart template.

1. Open one of your generated UIX files in the Visual Editor of JDeveloper.

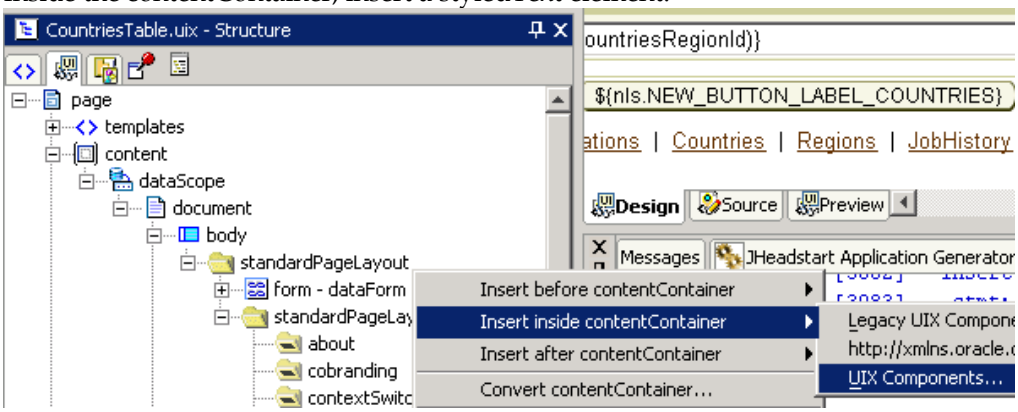
2. In the Structure Navigator, go to element 'standardPageLayout' and expand the node 'standardPageLayout named Children'



3. Insert a contentContainer inside the 'end' named child as follows:



4. Inside the contentContainer, insert a styledText element.



5. Enter "$GROUP_PROPERTY_1$" in the text property of the Styled Text element.

6. Go to the Source code of the UIX page and locate the code that's added in the previous steps. Look for:

```
<end>
  <contentContainer>
    <contents>
      <styledText text="$GROUP_PROPERTY_1$"/>
```

```
        </contents>
      </contentContainer>
    </end>
```

just above the line: `</templates:standardPageLayout>`.

You have now changed one of your generated pages by doing a post generation step. We will now make the same change to the JHeadstart template, so all our generated screens will get the new Look and Feel.

1. Copy JhsTemplates.jtp to JhsTemplatesCustom.jtp. Change the **Template Properties File** property for the group to JhsTemplatesCustom.jtp.
2. Open JhsTemplatesCustom.jtp. Change the line `dataPage=template/dataPage` to `dataPage=template/dataPageCustom`
3. Copy template dataPage.jut to dataPageCustom.jut (use jjt version when generating for JSP).
4. Open dataPageCustom.jut. Copy and paste the code fragment from your generated page to the dataPageCustom.jut just above the line: `</templates:standardPageLayout>`.
5. Type some text in the **Custom Property 1** of your JHeadstart base group.
6. Generate your application. The JAG places the text in the Custom Property 1 in the generated pages. You should have something like this:

You can use the custom property to influence what is generated. This gives extreme flexibility, you can add images, buttons, titles, text and so on. You only need to add $GROUP_PROPERTY_*$ placeholders in the right place in your template and fill in the properties in the JHeadstart Application Structure File Editor.

Expression Language in Custom Properties

You can even use EL expressions in properties:

1. Change **Custome Property 1** to Output of EL: ${nls.ADVANCED_SEARCH}
2. Regenerate the page. (Of course this example makes no sense, it is only meant to demonstrate the use of Expression Language. Take a look in the ApplicationResources.properties file to find out what happens).

You should get this:

Specifying the overall Look and Feel using UIX Templates

The general look of your application is already determined by the UIX look and feel, however you can use UIX templates to give your application a specific corporate branding.

The advantages of templating are two-fold:

1. The default Look and Feel is defined only once in templates and used throughout the application.
2. You can 'easily' implement your own look and feel by changing templates or replace them by new ones.

Attention: The basic idea of templating is quite easy. However, implementing a custom Look and Feel is not a trivial task. You need knowledge of JHeadstart AND of the View Technology used.

JHeadstart uses a number of standard template files:

- jheadstartLibrary.uit
  This is the overall template file that indicates which other template files should be used. These are the following:
- standardPageLayout.uit
  Every page will use this template file. It specifies the Global Buttons, Corporate Branding, Product Branding and Copyright Message.

To provide your application with your own look and feel (also known as skin) you can modify these template files or make your own using these standard template files as your starting point. If you create new templates, include them in the template library.

All template files, should be located in the UIX folder you have specified in the **JSP/UIX Common Virtual Directory** property on the service level. The standard JHeadstart template files are generated into this directory when using the JAG.

See the UIX Developers Guide (available in the JDeveloper online help) for more information about creating your own templates.

Reference: See 'How To create a Look and Feel for ADF UIX' on http://www.oracle.com/technology/products/jdev/howtos/10g/adf_uix_laf_ht/index.html for instructions on how to create and package your own skin.

Specifying the overall Look and Feel for your JSP application

The general look of your JSP application is determined by the Cascading Style Sheet specified in the Application Structure File, and some generic jsp include files.

JHeadstart is delivered with CSS and JSP include files. You can customize these to provide the look and feel of your own application. This chapter is not a tutorial on how to use cascading style sheets and JSP include files, but indicates how some easy, minor customizations can be performed in the files provided with JHeadstart.

Change the Corporate or Product Branding

The image files for the corporate and product branding are included in the header.jsp file in the jsp includes directory. Simply change the default gif files to your own gif files, and locate the gif files in the …/web/images directory.

Example: Change header image

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c"%>

<table id="header" width="100%">
  <tr>
    <td align="left">
      <img src="<c:out
value='${applicationScope.requestContextPath}/images/<your image here>'/>"
alt="Oracle" border="0">
    </td>
  </tr>
  <tr>
    <td align="left">
      <font class="header">  <application name here>/font>
    </td>
  </tr>
</table>
<br/>
```

Including Copyright

The copyright is included in the footer.jsp file in the jsp includes directory. Change this file as needed for your own application.

Example:

```
<table id="footer" width="100%" class="footer">
  <tr>
    <td align="left">
      <br/>
      &copy; <copy right message here>. All rights reserved.
    </td>
  </tr>
</table>
```

Internationalization

When generating your application, JHeadstart generates a resource bundle that holds translatable text. The name of the resource bundle can be specified in the Service-level property **NLS Resource Bundle**. Using the **Resource Bundle Type** property you can specify whether the resource bundle is generated as a property file, or as a java class.

Button labels, page header titles, and other fixed "boilerplate text" generated by JHeadstart are always generated into the resource bundle. However, if your application should be truly multi-lingual, meaning that the generated pages cannot contain hardcoded text at all, you should check the checkbox **Generate NLS-enabled prompts and tabs** as well. When checked, the prompts, tab names and display titles that you specify in the ADF BC property editor and Application Structure File Editor will also be generated into the resource bundle.

Internationalization	
* NLS Resource Bundle	view.ApplicationResources
* Resource Bundle Type	propertyFile
* Override NLS Resource Bundle Entries?	☑
* Generate NLS-enabled prompts and tabs?	☐
* Generator Default Locale	en
Generator Locales	nl, fr

In the **Generator Default Locale** property, you specify the locale that should be used to populate the default resource bundle, which is the bundle that does not have the locale suffixed to the name, for example `ApplicationResources.properties`. This resource bundle is used when the user's browser is set to a locale that is not supported by your application.

In the **Generator Locales** property, you can specify all other locales that must be supported by your application as a comma delimited list. For each locale in this property JHeadstart generates a resource bundle with the name as specified in the NLS Resource Bundle property, suffixed with the locale code, for example `ApplicationResources_nl.properties` and `ApplicationResources_fr.properties`.

Supported Locales

JHeadstart has built-in support for the following locales:

- pt_BR : Brazilian Portuguese
- hr: Croatian
- nl: Dutch
- el: Greek
- en: English
- fo: Faroese
- fr: French

- de: German
- ja: Japanese
- kr: Korean
- no: Norwegian
- sr: Serbian
- sl: Slovenian
- es: Spanish

Built-in support mean that if you specify one or more of these locales in the Application Structure File, the Resource Bundle generated for that locale will contain the correct translations for button labels, page titles and other fixed boilerplate text generated by JHeadstart. If you have checked the checkbox **Generate NLS-enabled prompts and tabs** then each resource bundle will also contain entries for the prompts, tab names and display titles, but these entries are still in the language you used to specify them in the ADF BC property editor and Application Structure File Editor. You will need to translate these entries manually in the generated resource bundle. After you have done this, make sure you uncheck the checkbox **Override NLS Resource Bundle Entries** to preserve your changes when you generate your application again.

Adding a non-supported Locale

If you want to generate your application using a locale that is not in the above list, you can do so by performing the following steps:

1. Specify the locale in either the **Generator Default Locale** property or in the **Generator Locales** property.
2. Generate the application.
3. Translate the entries in the generated Resource Bundle for your locale.
4. Uncheck checkbox **Override NLS Resource Bundles** in the Application Structure File, to preserve your translations. JHeadstart will then only add new keys, not change existing ones.
5. Modify <HTML Root Directory>\jheadstart\messages.js and add messages in your language.
6. Translate the JSP calendar (the UIX calendar already supports a large number of languages). See next section.
7. Add entries in your Resource Bundle for the JHS-messages (open jhsadfrt_source.zip and view the contents of the JhsUserMessages_<language>.java file for example messages).

Attention: The recommended type for Resource Bundle is java instead of propertiesfile if you have special characters in your language. Make sure you compile (rebuild) the Java Resource Bundle after you have added new entries, otherwise JHeadstart Application Generator will erase them the next time you run.

8. Make sure that your application users set the same locale in the browser and in their operating system (Windows: Control Panel – Regional Options). Some language dependent features (in particular UIX) use the browser locale, others the Windows locale.

Translating the JSP Calendar

The calendar provided with the jsp generation is a separate, third party product. Currently the calendar supports the languages mentioned above. If you want to add a new language, you need to change the javascript file `calendar.js` which can be found in the subdirectory `/web/jheadstart/calendar` of your project directory.

The month translations are specified in the function `getMonthSelect`. If you take a look at for instance the german part, you'll see the following part of code:

```
    // IF GERMAN
    else if (selectedLanguage == "de") {
        monthArray = new Array('Januar', 'Februar', 'März',
                     'April', 'Mai', 'Juni',
                              'Juli', 'August', 'September',
                     'Oktober', 'November', 'Dezember');
    }
```

For a new language you have to provide the same piece of code, with a different selectedLanguage and different phrases for the months of course. SelectedLanguage refers to the language setting of your system (in Javascript `navigator.language()` (Netscape) or `navigator.userLanguage` (IE)). The first two characters of the language string specify the language. The last three optional characters specify the language subtype, e.g. en_us. For a date format like "dd-mmm-yyyy", the calendar.js uses the first three letters of the complete month phrase for the "mmm" part.

The weekday translations are specified in the function `createWeekdayList`. For the German language, this looks like this:

```
    // IF GERMAN
    else if (selectedLanguage == "de") {
        weekdayList  = new Array('Sonntag', 'Montag', 'Dienstag',
'Mittwoch', 'Donnerstag', 'Freitag', 'Samstag');
    weekdayArray = new Array('So','Mo','Di','Mi','Do','Fr', 'Sa');
}
```

In the same way as you already have done for the months, just provide the same piece of code (with a different selectedLanguage and different phrases) for the new language.

The last thing you need to provide a translation for is for the word "Today". This can be done in the function `getTodayPhrase`.

Character encoding

JHeadstart uses a servlet filter to set the correct encoding on the HttpRequest and HttpResponse. By default, this filter sets the encoding based on the browser's locale setting. The mapping between locale and character encoding is the same as used by Oracle Containers for J2EE (OC4J):

- ar: ISO-8859-6
- be: ISO-8859-5
- bg: ISO-8859-5
- ca: ISO-8859-1
- cs: ISO-8859-2
- da: ISO-8859-1
- de: ISO-8859-1

- el: ISO-8859-7
- en: ISO-8859-1
- es: ISO-8859-1
- et: ISO-8859-4
- fi: ISO-8859-1
- fr: ISO-8859-1
- hr: ISO-8859-2
- hu: ISO-8859-2
- is: ISO-8859-1
- it: ISO-8859-1
- iw: ISO-8859-8
- ja: Shift-JIS
- ko: EUC-KR
- lt: ISO-8859-4
- lv: ISO-8859-4
- mk: ISO-8859-5
- nl: ISO-8859-1
- no: ISO-8859-1
- pl: ISO-8859-2
- pt: ISO-8859-1
- ro: ISO-8859-2
- ru: ISO-8859-5
- sh: ISO-8859-2
- sk: ISO-8859-2
- sl: ISO-8859-2
- sq: ISO-8859-2
- sr: ISO-8859-5
- sv: ISO-8859-1
- th: TIS-620
- tr: ISO-8859-9
- uk: ISO-8859-5
- zh: GB2312
- zh_HK: Big5
- zh_TW: Big5

If you want to specify a different encoding, perform the following steps (in the example we set the encoding to UTF-8):

1. add an init-param in the character encoding filter of your web.xml file as follows:

```
<filter>
   <filter-name>CharacterEncodingFilter</filter-name>
   <display-name>CharacterEncodingFilter</display-name>
   <filter-class>
      oracle.jheadstart.controller.CharacterEncodingFilter
    </filter-class>
   <init-param>
     <param-name>encoding</param-name>
     <param-value>UTF-8</param-value>
   </init-param>
 </filter>
```

2. Change the encoding of JDeveloper. Select 'Tools – Preferences' and select 'UTF-8' as encoding. JHeadstart uses the JDeveloper encoding setting for the generated Application Structure File. Also generated UIModels will get this encoding.
3. Change the encoding in the JHeadstart generator templates to UTF-8. See Using Generator Templates to find the location of the templates. The encoding of the templates gets generated into the generated pages.
4. Change the encoding in the Domain Definition file.

Suggestion: If your language contains special characters that are not properly shown in the resulting application, consider using Unicode notation. The tool native2ascii (see http://java.sun.com/j2se/1.3/docs/tooldocs/win32/native2ascii.html) can help you get the right Unicode for a specific text.

There's also a JDeveloper extension available at http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/native2ascii/index.html

Security

JHeadstart can generate security features into the application in two places: in the View layer by hiding tabs that the user is not authorized for, and in the Controller by using the standard Struts 'roles' property. This property prevents the execution of a Struts Action if the currently logged in user does not have access to any of the 'roles' listed in this property.

JHeadstart allows you to use JAAS or your own custom security mechanism, in a fully transparent way to the (generated) pages and struts-config files. How this works and you configure this is described in detail in Chapter "JHeadstart Extensions to ADF Runtime", in the "Security" section. For generating security into your application, however, you only need to know about two properties in the Application Structure File.

Restricting access to Groups

When some Groups in your Application Structure File should only be accessible to users that have certain privileges, you can use the Group level property "Authorized Roles/Function" to specify a comma-separated list of roles/functions that you want the logged in user to belong to/have access to if he wants to access any of the pages in this group. Note that he gets access if he belongs to one or more of these roles, not only if he belongs to all of them.

⊟ **Authorization**	
Authorized Roles/Functions	admin,manager

What JHeadstart will now do is generate the standard Struts 'roles' property on all Actions generated in the struts-config for this group. Struts will use this to check whether the logged in user has access to at least one of these roles, and if not, it would normally send an HTTP-400 error. JHeadstart has expanded this functionality and will forward to the 'accessDenied' forward (if present) instead. As a post-generation change, you can add such a forward to Actions to have fine-grained control over what should happen when the user is not authorized for that specific Action (or rather, the Group this Action belongs to). JHeadstart has added a global 'accessDenied' forward to the struts-config file, that points to a generic 'not authorized' page.

Preventing navigation to unauthorized pages

It's good to get an error page when someone tries to enter a Group he is not authorized for, but often it is much more user-friendly to prevent the user from navigating to places he isn't allowed to enter in the first place. This you can accomplish by setting the Service level property 'Hide Unauthorized Group Tabs'.

⊟ **UI Settings**	
* Show Hint Text?	☐
* Date Format	dd-MMM-yyyy
* DateTime Format	dd-MMM-yyyy HH:mm
* Set Date(time) Formatter on EntityObject Att...	☑
Default Display Width	60
* Allow Partial Last Page in ViewObject Page It...	☑
* Hide Unauthorized Group Tabs	☑
Read Only EL Expression (UIX Only)	

When this property is checked, JHeadstart will generate code into the UIX or JSP pages that prevents tabs, but also navigation buttons, from being displayed if they navigate to a group that the user is not authorized for. This is done using EL expressions such as this (UIX example):

```
<link text="Admin Pages" rendered="${jhsUserRoles['admin,manager']}" …/>
```

The jhsUserRoles object gives access to the authorization information. Details about this object can be found in Chapter "JHeadstart Extensions to ADF Runtime", in the "Security" section. For now it is only necessary to know that you can use similar EL expressions in your pages to anywhere something should be made dependent on the privileges of the logged in user.

Making field read-only based on authorization information

As we just described, JHeadstart allows you to access authorization information in EL expressions through the 'jhsUserRoles' object. One common usage of this is to make fields read-only based on authorization information for the currently logged in user. In fact this is so common, that we made it possible to generate such behaviour, through the following properties on the Entity Attribute or View Attribute level in the JHeadstart ADF BC Property Editors:

Display Settings	
Prompt	
Display?	
Display in Tables?	
Read Only in Forms?	${jhsUserRoles['admin,manager']}
Read Only in Tables?	${jhsUserRoles['admin,manager']}

And as the icing on the pudding, we acknowledged the fact that a common requirement is that not just individial fields, but an entire application should be rendered as read-only when the current user has a 'viewer' type role. For this purpose, we introduced the following Service level property in the Application Structure File Editor:

UI Settings	
* Show Hint Text?	☐
* Date Format	dd-MMM-yyyy
* DateTime Format	dd-MMM-yyyy HH:mm
* Set Date(time) Formatter on EntityObject Att…	☑
Default Display Width	60
* Allow Partial Last Page in ViewObject Page It…	☑
* Hide Unauthorized Group Tabs	☑
Read Only EL Expression (UIX Only)	jhsUserRoles['viewer']

Note that this expression should NOT be surrounded with the EL brackets ${…}, as it will be pasted in another, already existing EL expression.

What was generated for what purpose

This section describes the files you get in your project when generating with JHeadstart

When generating for the first time in a project

When generating, JHeadstart uses templates as a starting point. These files are only added to your project when they are not present. If they do not exist, JHeadstart adds them.

FileType	Location	Purpose
JhsTemplates.jtp	/properties	The JHeadstart Template Properties File lists the names and locations of the JHeadstart Template files
*.jjt and *.jut	/template	Templates used for generating the various page types. See Using Generator Templates

First and next time generation

During each generation, JHeadstart generates files for View and Model layer.

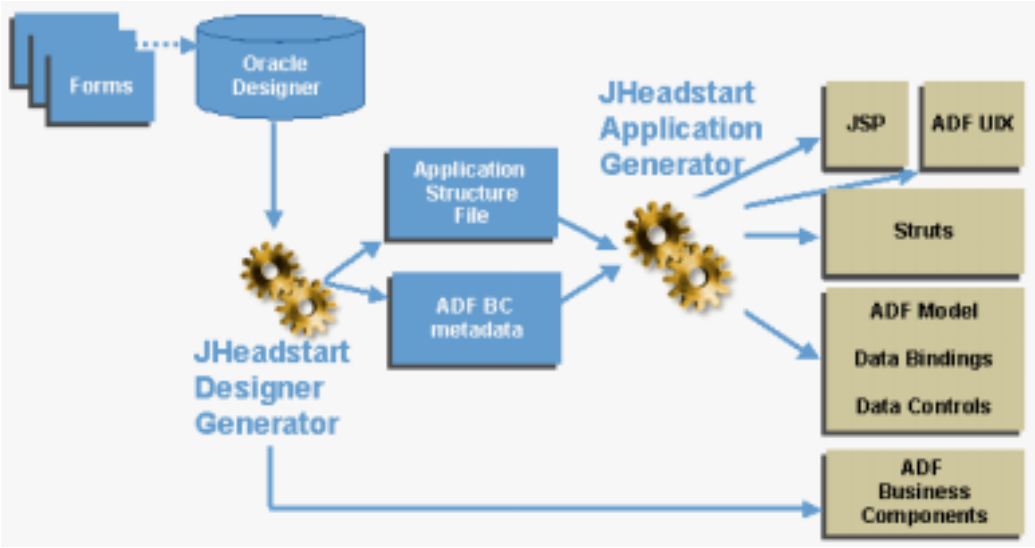
File Type / File	Location	When	Purpose
UIX pages	**UI Pages Virtual Directory** property at service level	**View Type** is UIX	UIX files define the application page using the User Interface XML.
UIT templates	/ **UI Pages Virtual Directory** property at service level	**View Type** is UIX	UIT files are templates used by the UIX technology. (Not to be confused with JHeadstart generator templates)
dateformat.properties	Java Source Path of JDeveloper project	Always	Date/Timestamp settings as defined in the **Date Format** and **DateTime Format** properties at the service level.
DataBindings.cpx	Java Source Path of JDeveloper project	Always	Container file for ADF Model layer.
*UIModel.xml	**View Base** property at service level	Always	Holds definition of ADF Data Bindings
NLS Resource Bundle	**NLS Resource Bundle** property at service level	Always	Resource Bundles that contain language dependent texts.
JSP Pages	**UI Pages Virtual Directory** property at service level	**View Type** is JSP	JSP files for the application pages.
JSP includes	**JSP/UIX Common Virtual Directory** property at service level	**View Type** is JSP	JSP files for the application pages.

CHAPTER

4

JHeadstart Designer Generator

The JHeadstart Designer Generator is an addin to JDeveloper that enables you to generate J2EE applications out of Oracle Designer based on meta data that is already available in the Oracle Designer Repository. In addition it allows you to migrate Oracle Forms to J2EE applications. You either do this straight from Oracle Designer, when you generated the Forms with the Designer Forms Generator or if you did not use Designer to generate your Forms you can Design Capture the Forms into Oracle Designer and run the JHeadstart Designer Generator (JDG). The JDG creates the Business Services (ADF Business Components) and the JHeadstart meta data. It can automatically call the JHeadstart Application Generator to generate the J2EE application that is fully based on Oracle ADF. Schematically the process looks as follows:



To generate the ADF Business Components the JDG actually uses the ADF Business Components generator that is shipped with Oracle JDeveloper.

In short, the ADF Business Components generator generates the following objects:

- Entity Objects
- Associations
- View Objects
- View Link Objects
- Application Module

The JHeadstart Designer Generator adds information needed by the JHeadstart Application to these objects and in addition generates the following:

- A domain definition file that contains the definitions of all static and dynamic enumerated domains
- An Application Structure file

The ADF objects created will have a collection of so called Custom Properties, representing additional properties of Entities, Views and Attributes over and above the standard ADF properties. The JHeadstart Application Generator uses these properties while building the View component of the application.

Later in this chapter, when we use the term JHeadstart Designer Generator (JDG), we talk about the JHeadstart Designer Generator including the ADF Business Components generator.

This document explains how to prepare your application in Oracle Designer to maximize the benefits of using the JHeadstart Designer Generator, as well as how to use the JHeadstart Designer Generator. An overview of how the definitions (elements and properties) in Designer translate to the objects and files generated by the JHeadstart Designer Generator, is provided in the JDG Reference.

Finally, the document explains the actual generated result to help you to check the result after having run the JHeadstart Designer Generator. This chapter refers to a Business Rule White Paper quite often. To not repeat the reference on every page the reference is shown here.

Reference: For information about enforcing business logic within the ADF BC Business Service, see the whitepaper 'Business Rules in ADF BC', which will be published on Oracle Technology Network (http://www.oracle.com/technology) soon.

This is the successor of the whitepaper BC4J Business Rules in BC4J: http://www.oracle.com/technology/products/jdev/htdocs/bc4j/BusinessRulesInBc4j.pdf

Introduction

The JHeadstart Designer Generator (JDG) can be used in different situations, but is probably of particular interest if you have used Oracle Designer as your development environment over the years, and now want to partly or fully migrate your application to a J2EE/JHeadstart application, or to develop some J2EE/JHeadstart add-ons to your existing application. In general, you can classify three situations where you may decide to use the JHeadstart Designer Generator (JDG).

- The major part of your application will be/is developed using the Oracle Forms generator, but you want to develop some self service of internet modules (HTML/J2EE) in addition to your existing application. You base your J2EE modules on the same data model, and mainly use the same display properties as you do in your Forms modules. The J2EE/JHeadstart modules are separate from the Forms modules. This will allow you to build your J2EE/JHeadstart completely independent of your Forms modules to maximize the use of the JDG.
- You want two different implementations of some of your modules, e.g. one Oracle Forms implementation, and one J2EE/JHeadstart implementation. You prefer to use the same module definition to prevent additional maintenance effort. This situation will require that any change you want to perform to the module to improve the J2EE/JHeadstart implementation will require more care, as you still have the requirement to generate a proper Forms module from the same module definition.
- You want to migrate modules from an Oracle Designer implementation (Forms and Reports) to a J2EE/JHeadstart implementation. There is no requirement to maintain the module definitions to generate the form/report after having migrated the application. In this situation you can make any changes to the module definitions to maximize the use of the JDG.

You may also be in a situation where a combination of these three alternatives is applicable.

Roadmap

When you plan to transform your Oracle Designer application into a JHeadstart Application, it is recommended that you follow the steps below:

1. Prepare your application in Oracle Designer

   The JHeadstart Designer Generator (JDG) uses many properties you already have defined for your server model objects and modules in Oracle Designer.

   If you use existing modules for the JDG generators, then you should be aware that some of the properties may not be processed as you may expect them to be, or may not be processed at all.
   Therefore, it is recommended that you walk through your Oracle Designer definitions to determine whether you need to make some changes, and to identify areas where you need to perform some post-processing steps.
   The section 'Prepare in Oracle Designer for generation' later in this chapter will help you in performing this step.

   If you want to use the same module for both Oracle Designer generation and JDG generation some of the properties may be conflicting. Therefore, JDG allows you to enter specific JDG hints to instruct the JDG to generate something else than the actual property value. These JDG hints are explained more in detail in the section 'Prepare in Oracle Designer for generation' later in this chapter.

   If you create completely new modules specifically for the JDG generator, you should learn which properties are of importance for the JDG generator. The section 'Create completely new JDG modules in Oracle Designer' later in this chapter will help you performing this step.

2. Prepare your project in JDeveloper

   Before you can run the JDG you must create and prepare your project in JDeveloper. See Chapter 2 '*Getting Started*' on how to prepare your JDeveloper project.

3. Run the JHeadstart Designer Generator

   You are now ready to actually run the JHeadstart Designer Generator. It is recommended that you start off with the simplest modules in your application to build experience with the generator and the generated result. See the section 'Generate Applications using the JHeadstart Designer Generator' in this chapter on how to start and use the JDG.

4. Check the generated result

   Walk through your generated result, the ADF Objects, the Application Structure file and the Domain file to verify whether the result is as you expected it to be. The section 'Understanding the outputs of the JHeadstart Designer Generator' will help you to see how the various Oracle Designer objects are transformed into ADF objects and files. The section 'Checking the generated result' will help you to check the actual generated result.

5. Generate and run the application using the JHeadstart Application Generator

   See chapter 3 '*JHeadstart Application Generator*' on how to perform these steps.

Understanding the outputs of the JHeadstart Designer Generator

The output of the JHeadstart Designer Generator (JDG) is the same as the input for the JHeadstart Application Generator (JAG) that is explained in detail in Chapter 3 *'JHeadstart Application Generator'*.

This chapter will give you a short overview of the output and explains how this can be related to Oracle Designer objects.

ADF Objects

The JDG creates a number of different ADF objects based on the server model and the modules in Oracle Designer.

ADF Objects created from Oracle Designer server model

Each table, view or snapshot (because of a usage from one of the modules selected for processing) is first processed into an **Entity Object (EO).** Each column in the table is processed into an Entity Object **Attribute**.

Primary key constraints are processed into Primary key attributes in the Entity Objects.

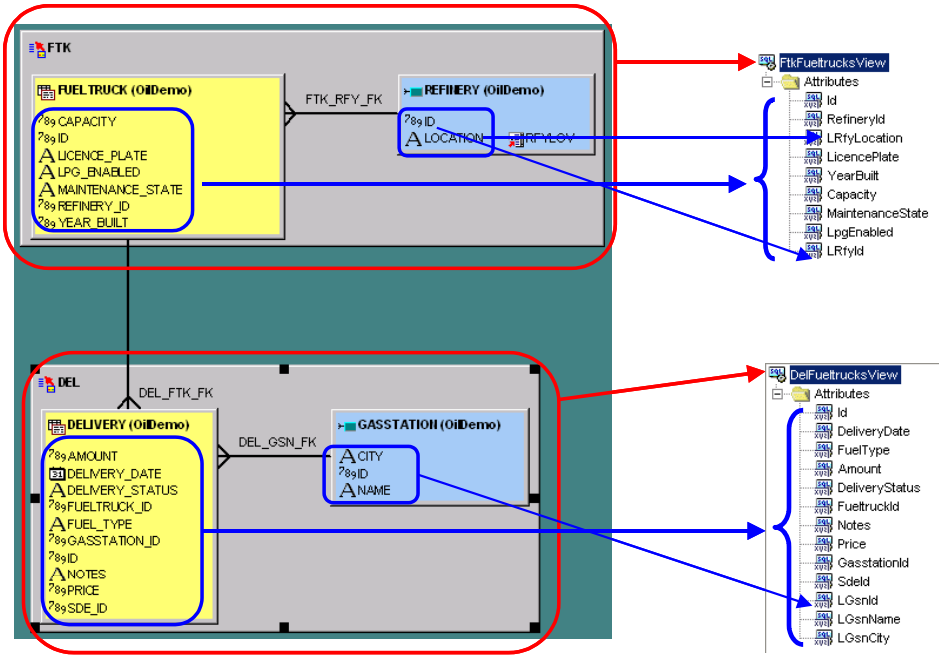
Foreign Key constraints are processed into **Associations** that are used for middle-or business logic tier implementation of the Foreign-Key constraints. These Associations are created between the two EO's created from the two tables between which the Foreign Key constraint lies. You can compare this to a client side implementation of a Foreign Key.

Primary key constraints are also processed into **Entity Constraints**. The ADF framework does not use these constraints at runtime.

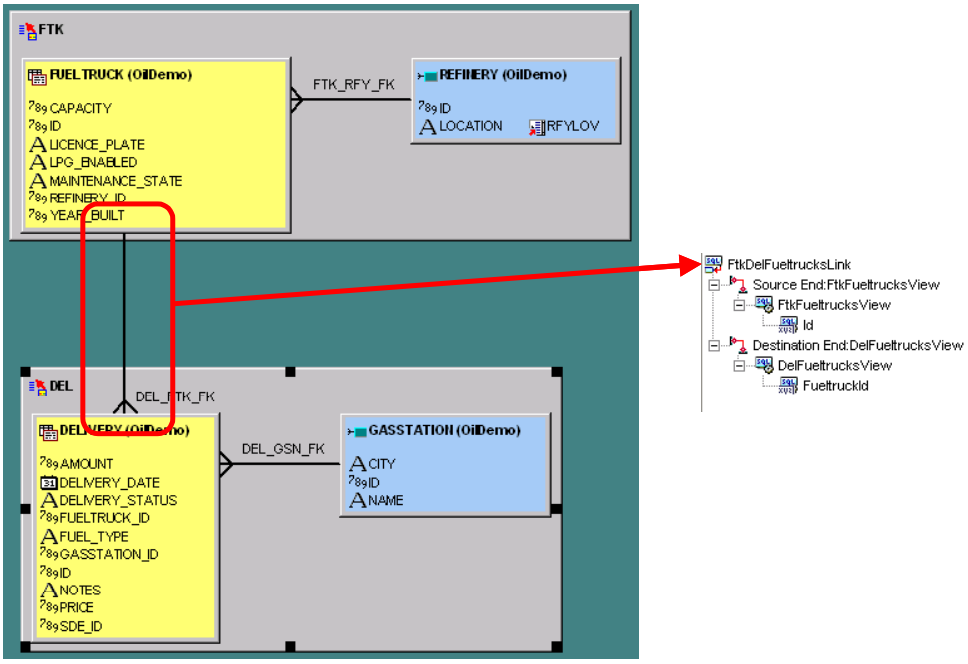
Finally, an **Application module** is created that serves as a container for all the created View Objects. All View Objects are included by the JDG in this Application Module.

ADF Objects created from Oracle Designer modules

For every module that is processed, one **View Object** is created for each Module Component and each List of Values used in the Module. For reusable module components and lists of values, there will be only one VO, regardless how often these reusable objects are used in various modules. This View Object contains attributes for all the bound items and certain unbound items belonging to the Module Components, including the lookups items in the Module Component. These VO's form the basis for the pages that are generated using the JHeadstart Application Generator.



View Link Objects are created for any Foreign Key based links defined between module components in the module.



See the JDG Reference for more detail about how each object and property in Oracle Designer is translated.

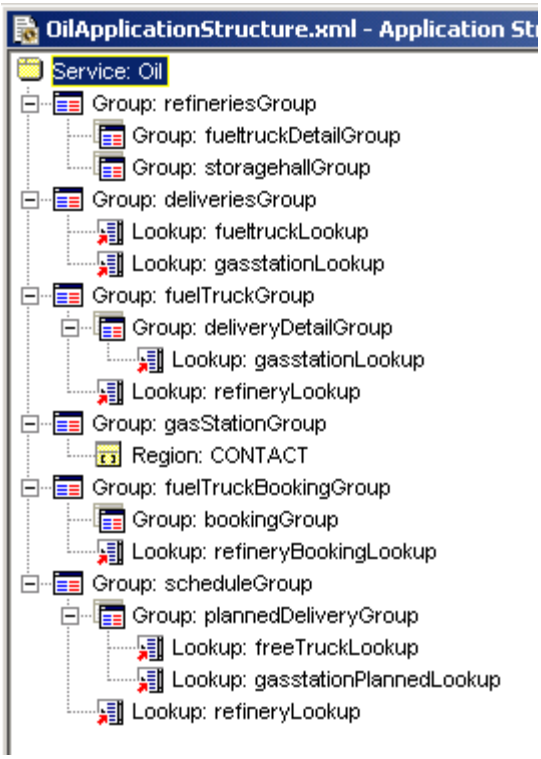
Application Structure File

The JDG also generates an Application Structure File (ASF). The Application Structure file contains a number of groups, sub-groups, lookups and regions.

The Application Structure File is a simple XML file that has the following overall structure:

```
<Service>
    <Group>
        <Lookup>
        </Lookup>
    </Group>
    <Group>
        (...)
        <Group>
            <Lookup>
            </Lookup>
            <Region>
            </Region>
        </Group>
    </Group>
</Service>
```

This structure shows two main groups, the first with one lookup, and the second with a sub-group that has a lookup on its own. The best way to view the Application Structure File is through the Application Structure File editor (see Chapter 3 for details on how to start up and use this editor):



The JDG generates a **Group** for each module component in a module. A main group is created for each module component (MC) that

- is not the child of another MC, or

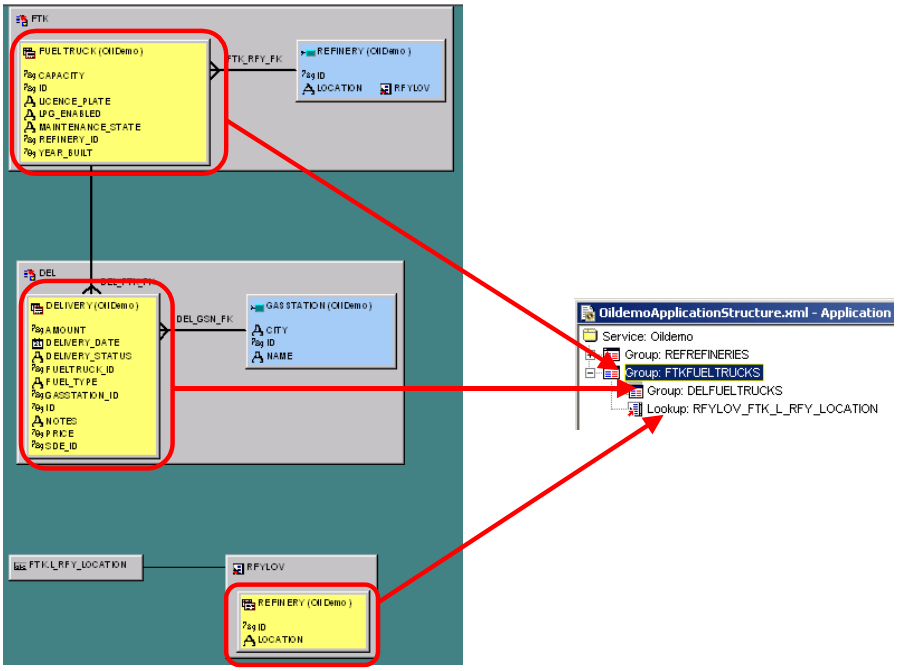
- is the parent of one or more MCs

A sub-group is created for each MC that

- is the child of another MC.

A **Lookup** is created for each List of Values inclusion.

A **Region** is created for every item group that contains bound items.



At the right hand side, we see the Application Structure File viewed through the Application Structure File Editor: Right-click on the Application Structure file in JDeveloper and select the Application Structure Editor.

Domain Definition File

The domain definition file contains all the enumerated domain definitions you will need in your application. For example a domain maintenance_state with 3 allowable values may be processed into the following:

```
<DomainSet>
   <Domain name="MAINTENANCE_STATE">
      <AllowableValue value="1" meaning="Excellent"/>
      <AllowableValue value="2" meaning="Average"/>
      <AllowableValue value="3" meaning="Terrible"/>
   </Domain>
</DomainSet>
```

The JAG uses this file to generate static domain definitions into the View Components of your application. Note that at this point, all dynamic domains that would normally be generated into CG_REF_CODES (or a user defined table) are also included as static domains in this file.

Create completely new JDG modules in Oracle Designer

If you already use Oracle Designer in your development environment and you want to develop some J2EE/JHeadstart modules in addition to your existing application, you may decide to create JHeadstart modules in Oracle Designer and use the JDG to create the basis for the JHeadstart Application Generator. This is particular of interest if you base your J2EE modules on the same data model, and mainly use the same display properties as you do in your Forms modules.

This section is not a tutorial on how to create modules in Oracle Designer, nor does it cover a detailed description for every property you can set. This section focuses on the steps you have to go through to provide a sound basis to create the modules in Oracle Designer, and covers some specific aspects you should be aware of when creating the modules. If you want detailed information on how each module property is handled by the JDG, see the JDG Reference Guide.

Check existing basis (domains and server model)

Before you start to create your JDG modules, you should check the domains that your JDG modules will use and the part of the server model on which you want to base your JDG modules. See the section 'Check domains' below what you need to consider if your JDG modules will use any of the existing domain definitions. See the sections 'Check Tables/Views/Snapshots and their columns' and Checking Keys for Tables/Views/Snapshots (Primary, Unique & Foreign Keys) to see what you need to check for the server model objects you will use in your JDG modules.

Creating your JDG modules

Create your JDG module as you would normally create a straightforward Oracle Forms module in Oracle Designer. The module works only as a framework for one or more module components as it is the module components that control the JDG generation. Set the language to Oracle Forms.

Creating your Groups via Module Components (MCO)

Create your module components, with table and lookup table usages with all the bound items you want to use in the final J2EE/JHeadstart pages. You can also use unbound items, but do not use any that are used specifically on the client, such as buttons, or expressions retrieved on the client. This must be handled afterwards in the J2EE application.

For all sub-Groups, create a new module component, and ensure that you create a key-based link between the master MCO (master group) and the detail MCO.

Forcing the Group Layout through Regions

If you want to split attributes of the Group into one or more Regions, then create an Item Group for each Region. Ensure that all the items that should be displayed within one region are located within the same Item group, and that the order of the items within the item group is appropriate, as well as the order of the Item Groups.

Creating your Lookups via List of Values (LOV)

For each lookup you need on your page, create a specific list of values (LOV) or include a reusable LOV in the module. Specify the item from which the LOV should be activated, and indicate through the return list which LOV item should be returned to which MCO item. Do not use more than two LOV items.

Handling Business Logic and Business Rules

It is recommended that you handle all your business rules in the database. If you have used CDM RuleFrame for your existing application, then your J2EE/JHeadstart application can be CDM RuleFrame enabled. This approach prevents you from having to re-implement the rules using ADF.

Any Business Logic that is not implemented in the database must be implemented using ADF. You should not enter any Application Logic for the Oracle Designer module. For more information, see *the Business Rule White Paper*.

Prepare in Oracle Designer for generation

To maximize the benefit of the JHeadstart Designer Generator, and to be prepared for any post-generations to perform, it is recommended to perform some preparations and checks in Oracle Designer prior to running the generator. This chapter indicates per Oracle Designer object which preparations are useful when using the JHeadstart Designer Generator.

Check domains

Enumerated domains are generated into the Domain Definition File used by the JHeadstart Application Generator. In addition they are generated as ListValidators for the Entity Object attributes using that domain. These are currently generated as ListValidators with Literal Values independent of whether the domain is dynamic or not. Therefore you have to pay special attention if you have used dynamic domains. These are the domains where the Dynamic List? property is set to Yes. For these you need to modify the generated ListValidator to use a Query Result instead of Literal Values. Also, be aware that the enumerated dynamic domains are included in the domain definition file when using the JDG. This means that they will be handled as if they were static. Walk through the domains in the Repository Object Navigator (RON) and identify all dynamic domains.

Range domains are generated into RangeValidators, with the minimum and maximum values indicated for the domain.

There is also a third type of domain that normally is used to provide a common format to various attributes and columns in Oracle Designer. These would not have any allowable values. These types of domains are ignored by the JDG as the format is generated based on the given format for the column.

Attention: You can run the script CheckDomains.sql to help you identify any domains or domain usages you have to take care of.

Using JDG Hints

When you use the same objects both for Oracle Designer generation (Server Generator, Oracle Forms generator, etc), and the JHeadstart Designer generator, there may be situations where the property values will be conflicting dependent on which generator uses the object.

To overcome this problem, the JHeadstart Designer Generator allows you to specify specific JDG hints to instruct the JDG to generate something different than specified. This prevents that you either have to make duplicate modules, or to switch the properties back and forth depending on which generator you use.

In this way you can influence the of properties at Application Structure File level or in the ADF Entity Object/View Object custom properties as follows:

Include in the **Notes** property (in Designer) of the
Column,
the (Databound) Item,

the Module Component,
the LOV Inclusion or
the LOV (one or more of) the following tag:

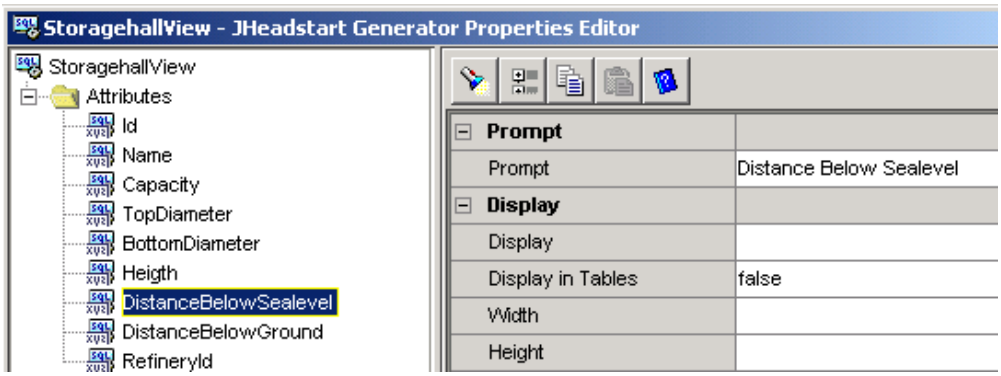
```
<JDG property="value" property="value" property="value"/>

For example for a MCO:
<JDG displayTitle="my displaytitle" useTableRange="false"/>
```

The JDG will pick up these hints and they will override the default derivation by the JDG. Furthermore, you can generate properties from Designer that cannot be derived from Designer, such as useTableRange and sortable. When you use such JDG hints, it is important that you use the property name of the actual Application Structure File xml, and the view object xml.

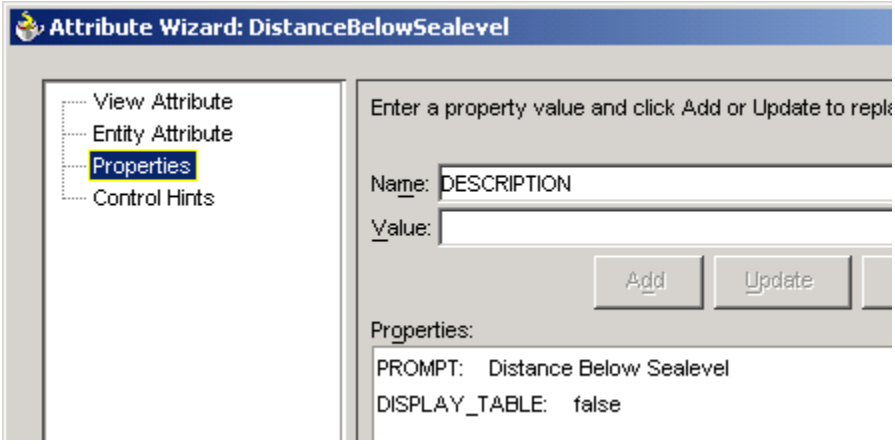
For the objects that transform into the Application Structure File (such as Module Components, List of Values etc), you can find the exact name by opening the Application Structure File using the xml editor instead of using the Application Structure File editor. Naturally, you must first provide a value for that value using the Application Structure File editor so that the property will appear in the file.

For the objects that transform into Entity Objects and View Objects (such as Tables, columns, bound items etc), you can find the exact name by opening the view or entity object using the Attribute wizard, and navigate to the Properties node. There, you can see the actual name of each custom property entered for that attribute. This means that you first have to enter a value for that property using the JHeadstart BC4J Property Editor to make it appear here.

Example: DistanceBelowSealevel seen through the JHeadstart ADF BC Property Editor:



DistanceBelowSealevel seen through the Attribute wizard.



As you can see, the Prompt property is called PROMPT, and the Display Table property is called DISPLAY_TABLE.

Check Tables/Views/Snapshots and their columns

This chapter explains the parts where you may need to do some preparations to Table/View/Snapshot definitions in Oracle Designer prior to generation, or where you need to make adjustments after having used the JDG. If you want to know in detail which properties in tables, views and snapshots are used by the JDG generator and how they are translated into the corresponding JDeveloper Objects, see the JDG Reference.

In the sections below, when we talk about tables, we also mean views and snapshots.

Below, each property of importance is discussed. The order of the properties is as they are represented in the property pallet in Oracle Designer.

Table - Name, Column - Name

The table name is used to form the name of the Entity Object (EO), and the column names to form the EO Attribute names. There are some more restrictions in how you can name an EO and its attributes compared to the table and columns names as they can be created in the database. You can enter names in the database containing the characters # and . but it is not allowed as a part of a ADF Business Component name. The JDG transforms the # and . characters into underscores, but keeps the underlying column and table name as it is.

Attention: A check to identify table and column names having these characters in the name is included in the script CheckTables.sql.

Column properties - Datatype

The JDG does not handle all the values Oracle Designer provides for the Datatype. If you have defined unusual datatypes for any columns, then these may not be processed as expected. See the JDG Reference for more details on how each datatype is processed by the JDG.

Attention: A check to identify usages of these kinds of unusual datatypes is included in the script CheckTables.sql.

Column properties – Uppercase

If you have specified that column values should be in uppercase the JDG generates a value 'upper' in the custom property Uppercase/Lowercase (RESTRICTED_CASE). The JHeadstart Application Generator does currently not use this property, so this will have no effect on your final generated application. Therefore, if this is an important functionality you will need to implement this separately after having run the JDG, and the JAG.

Attention: A check to identify uppercase columns is included in the script CheckTables.sql.

Column properties – Default Value/Default Value Type

It is the Default Value Type that determines how the default is translated by the JDG. If the Default Value Type is either Database Function Call or Client Function Call, then JDG generates the default value into a COMPLEX_DEFAULT custom property for the EO attribute. The JAG does not use COMPLEX_DEFAULT. It is there only to indicate to the developer that the default needs to be implemented as a post-generation step.

Currently the default is also generated into the default property of the EO Attribute independent of the kind of default (problem report: 3752253). Therefore, if the default is non-literal it must be removed after generation.

If the default is not derived in the server, e.g. through the TAPI, then you can code the default using the approach described in the *Business Rules White Paper*, section 'Change Event Rules with DML' subsection 'Default Rules'.

Attention: A check to identify non-literal defaults is included in the script CheckTables.sql.

Column properties – Sequence

The JHeadstart Designer Generator does not take into account any sequences that populate columns, however, if you have set the Server Derived property to Yes then the JDG ensures that the ADF Entity Object Attribute properties Refresh - After Insert and After Update are set to true, and sets Mandatory to false. This ensures that the end user will not be forced to enter a value, and that whenever the user has entered a new record, that the information on the page is refreshed with the server-derived sequence.

However, if you do not server derive the sequence value, then you have to write your own ADF code to ensure that the value is populated. See the *Business Rules White Paper*, section 'Change Event Rules' subsection 'Default Rules' for more information about how to do this.

Attention: A check to identify columns using sequences is included in the script CheckTables.sql.

Column Properties – Denormalization

This property is not processed by the JDG. If the column is Server Derived, then it is assumed that it will be handled by the TAPI. However, if the column is not Server Derived, you will have to manually add a Derivation rule to your Entity Object to perform the denormalization.
See the *Business Rules White Paper*, section 'Change Event Rules with DML' subsection 'Derivation rules' for more detail about how to do this.

Attention: A check to identify columns with denormalizations is included in the script CheckTables.sql.

Column Properties – Derivation – AutoGen Type

If you have used any of the **AutoGen Types** used to record history data, then the JDG checks the History Column indicator for the attribute and specifies the corresponding type. See the JDG Reference for more detail.
Also, if the Server Derived property is set to Yes, the JDG will set the ADF attribute properties Refresh – After Insert and After Update.

However, you should carefully consider how to handle this kind of history recording. Setting the value for CreatedBy and LastUpdatedBy on the server is not generally a good idea in web applications. Typically, the user logs into the application with a username that is –not- a database user, but is an application user. The application validates the user and connects to the database using a single 'super' username for all users. So, on the database, everyone has the same username. Thus, recording the database user in CreatedBy and LastUpdatedBy is meaningless. You want the application user.

Therefore, you probably want to change Server Derived to No for the CreatedBy and LastUpdatedBy columns.

Also, view the JDeveloper help on how the History Columns are handled. Finally, view the section 'Populating Audit Columns' in the *Business Rules White Paper*.

Attention: A check to identify columns with auto generated values is included in the script CheckTables.sql.

Column Properties – Derivation – Derivation Expression

The JDG ignores any derivation expression. If your derivation expression is not server derived, you should consider whether it is possible to derive the value in the server. If this is not an option, then you will need to add ADF code to ensure that the columns will be populated. See the *Business Rules White Paper*, section 'Change Event Rules with DML', subsection 'Derivation Rules' for more information about how to implement Derivation rules.

Attention: A check to identify columns with derivation expressions is included in the script CheckTables.sql.

Checking Keys for Tables/Views/Snapshots (Primary, Unique & Foreign Keys)

ADF does not provide a means to enforce Unique Key constraints within the ADF framework. Normally, unique keys are enforced only on the database. If you like, you can add business logic to the Entity Object to enforce unique keys within ADF. See the *Business Rules White Paper* for information about how to add business logic to an Entity Object.

Primary Keys

ADF requires that all Entity Objects be created with a Primary Key component. This means that if you have any tables or views in your repository used in module components, you must ensure that these have a primary key so that the primary key is handled by the JDG. If you have primary keys with the Complete property set to No, then these will be handled in the same way as the other primary keys. Therefore, it is recommended that you check these primary keys to verify whether any updates are required.

Attention: A check to identify tables with no primary keys, or with primary keys with the complete property set to No is included in the script CheckTables.sql.

Composite Primary Keys

If you use composite primary keys you must be aware that you cannot use the select-form layout option as you can only choose one of the key components to select the record. This will result in an ArrayIndexOutOfBoundsException when pressing the Edit button in the select page.

You may want to try to use a descriptor attribute containing the complete composite primary key, but this will also result in an error.

Therefore, it is recommended that you use a form layout instead, starting with a Find page, or that you introduce a new surrogate primary key consisting of a single column.

Attention: A check to identify composite primary, unique and foreign keys is included in the script CheckKeys.sql.

Foreign Keys

Foreign Keys defined as Cascade Delete in ADF result in Composite Associations. Such composite associations present us with problems when we try to query a ViewObject based on an EO on the detail (destination) end of such a composite association OUTSIDE the context of the master EO. For example, if the foreign key between EMP and DEPT is defined as Cascade Delete, the Association EmpDeptAssoc will be composite; as a result, the VO EmpView can never be queried outside of the context of a specific Department. This may be a reason to not set Foreign Keys to Cascade Delete

Mandatory Keys

In ADF all attributes that are part of a Mandatory Key are automatically set to mandatory. That means that if the Primary Key is mandatory, the attribute(s) that are part of this key are made mandatory as well. However, if the value of an attribute is server derived (for example through a sequence-based derivation in the Table API), the

EO attribute should not be mandatory, otherwise perfectly correct insert operations will fail in the middle tier. That in turn means that the Primary Key cannot be mandatory either. Therefore the JDG currently generates all Primary Keys in ADF as non-mandatory. Attributes in the Primary Key are made optional only when they are server derived or server defaulted.

Keys with no columns

Normally, you would not use primary, foreign or unique keys with no columns. The JDG ends with an error when there are no columns for primary and foreign keys. It is therefore necessary to ensure that all your primary and foreign have specified columns. Unique keys are not processed.

Attention: A check to identify primary, unique and foreign keys with no columns is included in the script CheckKeys.sql.

Preparing check constraints

Check constraints are not processed by the JDG. Server or Both Validated Check Constraints could have been processed into Entity Keys of type Check, but as these are meant for forward generation into the database, these are ignored. They would not have had an impact on the ADF runtime behavior.

If the constraint is set to Implement In Client, then you will need to manually add code to the ADF Entity Object to enforce the constraint. See the *Business Rules White Paper* for more information about how to implement Attribute and Instance rules in ADF.

If the constraint is set to Implement In Both, then you need to evaluate whether or not the constraint should be implemented in ADF, and if so, implement it manually.

Attention: A check to identify constraints that are only implemented on the client is included in the script CheckKeys.sql.

Check modules and module components

Using the JDG generator you can transfer many kind of modules. It is mainly the Module Components (MCO) within a module that controls the transformation of modules.

Each Module Component is processed into a ADF View Object that contains Attributes processed from the items in the MCO. In addition, a Group definition in the Application Structure File is created for each Module Component.

While processing modules, the JDG also processes the tables, view or snapshots that the MCO's belonging to the module use. See the section 'Check Tables/Views/Snapshots and their columns' above and the JDG Reference for details on how these are processed.

Also, all domains that are used by the selected modules will be processed by the JDG. See the section 'Check domains' earlier in this chapter, and the JDG Reference for details on how the domains are processed.

This chapter explains the parts where you may need to do some preparations to your module definitions in Oracle Designer prior to generation, or where you need to make

adjustments after having used the JDG. If you want to know in detail which properties in the various module objects are used by the JDG generator and how they are translated into the corresponding JDeveloper Objects, see the JDG Reference.

First, some information is provided about specific objects, or constructions you may have used in your module. Thereafter, each property of importance for the different module objects (such as module components, list of values etc) is discussed. The order of the properties is as they are represented in the property pallet in Oracle Designer.

Module networks

Module Networks are ignored by the JDG. If you have created module networks to call one form from another form using parameters, and you want a similar behavior in your java application, then you will have to perform modifications to the code after having used the JAG. See the UIX Developers Guide on how to create buttons for navigation.

Attention: A check to module networks - with the exception of library usages - is included in the script CheckModules.sql.

Module Components based on View API (VAPI)

If you have a module component based on a View API with an Instead Of trigger, and if that View API has server-derived columns (sequence, autogen), you will run into a limitation of ADF. For server-derived columns, ADF uses the Refresh After Insert and Update to generate a 'RETURNS' clause into the corresponding SQL insert and update statement. Unfortunately, it is not legal to use a RETURNS clause on a complex database view.
Therefore, when generating with the JDG, you cannot have server-derived columns on a View API.

Attention: A check to identify module components based on View API's with server-derived columns is included in the script CheckMCOs.sql.

Module – Short Name

The module short name is used to form the name of the generated ADF View Objects. There is a restriction on what kind of characters to name the ADF View Object. For example, characters like & % # - + = @ cannot be used to name a few. If you have used any of these or any other unusual characters you might get an exception when running the JDG, because the View Object cannot be created.

Attention: A check to identify modules with not usable characters in the name is included in the script CheckModules.sql.

Action Items

Action items are ignored by the JDG. This means that if you have used any of these, you will need to investigate their purpose to determine what your course of action will be.

If you have used a Navigation Action Item, then view the UIX developer's guide on how to implement navigation buttons in your new environment. If the button is not used for navigation, but to perform some kind of other action you will need to write your own method and include the button in your uix page.

Unbound Items

Unbound items are handled differently dependent of what type of unbound item it is.

If the unbound item is a Computed summary item a separate View Object is created with the summary item and its expression. This VO is however not included as a GROUP in the application structure file. You need to determine how (and if) you want to use this View Object.

The rest of the unbound items that are migrated are migrated as part of the VO created for the Module Component. Unbound items defined as Custom, Client Side function, SQL Aggregate or Target Specific are not migrated.

You need to determine the course of action for the unbound items that are not migrated. For some of these you should be able to achieve similar functionality in your J2EE application.

For example, you may have defined your unbound item as an image item. If you used Headstart you would have recorded the image you wanted to display between <IM></IM> tags in the hint text, or as a forms parameter.

Lets say you have one image item with the following in the hint text <IM>myimage.gif</IM>. If you still want to use the image, then you must include the image at the appropriate location in your generated uix file. See the example given for 'Images used for the wizard' in the 'Check Headstart Specific Constructions' below.

Attention: Checks to identify unbound items (buttons, images and other types of unbound items), and action items are included in the script CheckMCOs.sql.

Bound Items – Display Type

The Display Type for the bound item is processed into the custom property Display Type (DISPLAY_TYPE) for the corresponding VO attribute.

The following Oracle Designer display types will be translated to the following JHeadstart display types:

Oracle Designer Display Type	JHeadstart Display Type
Text	editor : if the maximum length of the column is larger than 80 characters dateField : if it is a date textInput in all other cases.
Check box	checkBox
Combo box	<none>
Descriptive Flexfield	<none>
Image	<none>

Oracle Designer Display Type	JHeadstart Display Type
OLE Container	<none>
Poplist	choice
Radio Group (Across)	radioAccross
Radio Group (Down)	radioDown
Text List	<none>
OCX Control	<none>
Display Item	displayField dateField : if it is a date

If you use any of the display types not supported by JDG, then you should consider which display type is appropriate.

If you will need the module for other generators than the JDG then you can add a JDG hint with the appropriate display type in the Notes property of the bound item as follows:

```
<JDG DISPLAY_TYPE="your display type"/>
```

Attention: A check to identify items with not supported display types is included in the script CheckMCOs.sql.

Bound Items – Show Meaning

The Show Meaning property will have no effect, as it is the meaning that always will be displayed when using domains.

If you use a static domain, and you need to display something else than the meaning then you can choose one of the following alternatives:

Type of Change	How	When appropriate
Change existing domain in Oracle Designer	Change the meaning in the domain to contain the value, as it should be displayed. For example, if Show Meaning has been set to Meaning alongside Code, then in the domain, enter the code and the meaning as the meaning for the domain.	If the domain is always displayed in the same manner, and the modules are not regenerated using Oracle Designer generators (e.g. OFG).

Type of Change	How	When appropriate
Create new domain in Oracle Designer	Copy the domain that is used and change the meaning as appropriate. Ensure that the column that used the old domain now use this new domain.	If the domain is displayed differently in different situations when used for different columns. If the domain is displayed different in bound items based on the same column, then this approach will not be appropriate as the domain is linked to the column, and not to a bound item. Also, this approach will not be appropriate if the modules are regenerated using Oracle Designer generators (e.g. OFG).
Change domain in generated domain file in JDeveloper	Change the meaning in the domain to contain the value, as it should be displayed on the page.	If the domain is always displayed in the same manner, but modules may be regenerated using Oracle Designer generators (e.g. OFG). Therefore the domain could not be changed in Oracle Designer.
Add domain in generated domain file in JDeveloper	Copy the domain definition from the generated domain. Change the meaning in the domain to contain the value, as it should be displayed on the page. Change the user defined property DOMAIN for all VO attributes that should use the new domain.	If the domain only should be used in specific situations.

Attention: A check to identify items with Show Meaning set to something else than Meaning Only (or null) is included in the script CheckMCOs.sql.

Bound Items – Default Value & Default Value Type

If you have defined a default for a bound item, then the value is generated into the default property of the VO attribute independent of the default type. Only the LITERAL defaults should have been generated as such so you will need to remove the other kind of defaults after generation (see problem report: 3752253).

In addition to this, the JDG generates the custom property COMPLEX_DEFAULT with the default value when it is not LITERAL. The JAG does not use COMPLEX_DEFAULT. It is there only to indicate to the developer that the default needs to be implemented as a post-generation step.

See the *Business Rules White Paper*, section 'Change Event Rules' subsection 'Default Rules' for more information on how to implement defaults.

Attention: A check to identify items with defaults is included in the script CheckMCOs.sql.

Check LOV's

The JDG transforms LOV's into an ADF View Object and a Lookup definition in the Application Structure File. You have to consider the following LOV properties when reviewing your LOV's.

List of Values - Return List

The JDG can handle a Return List up until two return list items. If you have included more return list items, then the lookupValueAttribute and the baseValueAttributes that are set in the Lookup definition in the Application Structure File will be confused. There can only be *one* displayed attribute, so therefore a long return list as you used it in forms generation will not be useful.

You can use the JDG hints to indicate to the JDG what should be the correct lookupValueAttribute and baseValueAttribute. For example, if you have a return list like

```
BV_PAS.PRJ_ID=PROJ_LOV.ID,
BV_PAS.PRJ_NAME=PROJ_LOV.NAME,
BV_PAS.PRJ_PRJ_TYPE=PROJ_LOV.PRJ_TYPE
```

and you want the id attributes to be the value attributes and the name attributes to be the display attributes, then you can include the following JDG hint in the Notes property of the LOV inclusion:

```
<JDG baseValueAttribute="PrjId" lookupValueAttribute="Id"
baseDisplayAttribute="PrjName" lookupDisplayAttribute="Name"/>
```

On the other hand, having more than two return list items implies that you have lookup items that you want to display with lookup values. The JDG has included your lookups as part of the View Object, so when you query the records the values will be displayed on the screen, but they will not be populated when using the LOV. You may want to consider using transient attributes to accomplish the desired behavior. See Chapter 3, '*JHeadstart Application Generator*', for more details about how to do this.

Attention: A check to identify LOV's with return lists having more than two return list items is included in the script CheckLOVs.sql.

List of Values - Additional Restriction

The additional restriction for the LOV is included in the where clause of the generated VO. This is independent of the Used For property. If the LOV is only used when entering data, then you should remove the restriction after generation. If the designer setting specifies that it should only be used for Query, then you should determine what to do. If the LOV should be used in a search page/area, you should keep the additional restriction. However, in many situations the LOV is used both in the search page/area and in the Data Entry page. You need to determine the best approach in these situations.

Attention: A check to identify LOV's with Additional Restrictions used for Query only or Data Entry only is included in the script CheckLOVs.sql.

Investigate application logic

The JHeadstart Designer Generator does not process application logic used at any level in your module. Therefore, you first of all have to determine whether you have used any Application Logic.

If you have used Application Logic in your modules, then you need to determine what is the purpose of the Application Logic:

- Is it a client implementation of a business rule? If so, will you need a similar implementation in your JHeadstart application? If so, you can implement the business rule as described in the *Business Rules White Paper*, or if you need faster user feed back, in JavaScript. Take into consideration that the rule may have been implemented in the database as well, so you might not have to implement the rule in the application anymore.
- Is it a specific handling related to the tool in which the module is implemented, e.g. Oracle Forms? If so, will you need a similar handling in your new framework?
- Is it an implementation of business logic covering a business requirement for the module itself? This is probably the most difficult type of application logic to transform. Take notice at which events the Application Logic is triggered, and try to determine how this can be translated to your new environment.
  Also, see if it is possible to reuse some code, for example by placing (parts of) the code in the database.

When you have performed an analysis of what kind of Application Logic you have used in your application, then determine a common approach on how to transform this into your new environment. Also determine a mechanism on how to keep track of the transformations you perform.

Attention: Checks to identify application logic on module component and item level are included in the script CheckMCOs.sql. A check to identify application logic on module level is included in the script CheckModules.sql.

Check Headstart specific Constructions

If you have used any Headstart specific constructions in Oracle Designer, such as Query Find windows, Dynamic Window Titling, etc, some of these are handled and some are ignored by the JDG. Therefore, you need to make considerations if you have used any of the following:

- Query Find windows

  Module Components created as Query Find windows normally use the following naming convention QF_<result_block>. It is expected that when you have used a Query Find window in your Oracle Forms module, you would like a Find Page

(Advanced Search on separate page) for the corresponding Page in your new application. Therefore, the JDG specifies that an Advanced Search page should be generated on a separate page. All the displayed items in the query find module components are made queryable in the VO that will be used.

- Control block generation

  Control block generation was achieved by using the template/library object QMSSO$CTRL_BLOCK or AQMSO$CTRL_BLOCK dependent on the Headstart flavor you used.

  If your Module Component has no table usage and is only based on unbound items, then the Module Component currently results in a View Object without any attributes. You need to determine the purpose of your Control Block, and decide how to handle dependent on this. If you need a similar functionality in your new application, you may be able to achieve this by adding attributes to the appropriate View Objects, and specify that they should not be selected in Query.

  If your Module Component is based on bound items, but through Headstart functionality work as a control block, then the JDG ensures that the control block is not generated. However, if the control block was used to generated Query Find functionality, then it has been processed as indicated for Query Find windows above.

  If however, the control block was used for any other purpose you have to - as described above - determine the purpose of your Control block, and decide how to handle.

Attention: A check to identify control blocks as described above is included in the script CheckMCOs.sql.

- Dynamic Window Titling and Positioning

  Headstart Oracle Designer and Headstart for Oracle Applications supports an easy mechanism to provide dynamic context sensitive window titling and positioning using html like tags <WT></WT> and <WP></WP> in the window title. JDG uses the Window Title when there is no title provided for the module component, removes the html like tags, and ignores the content used for Dynamic Window Titling or Positioning.

  The Positioning information is not useful in your new application. Each new page is located at the same position. Therefore you should remove any information provided between <WP></WP> in the window titles prior to generation. If the module may be regenerated using the Oracle Forms Generator, then you should not remove the tags. Instead, add a proper title for the module component as if there is a title provided here this is used instead of the window title.

  The window titling however was a means to show context information from the master block in the detail window title. When using the JDG, the column that is generated as the descriptor attribute will be shown as context information in the detail page. Therefore, if there is only reference to one bound item in the window title, then set for this bound item the Descriptor Sequence property to 1 to indicate that this should be used as the descriptor attribute.
  If there is a reference to more bound items, then you will need to perform a post-

JDG generation step and create a Logical View Descriptor in the View Object, and use this as the descriptor attribute in the Group Definition instead.

Attention: A check to identify Headstart typical dynamic window titling and positioning is included in the script CheckMCOs.sql.

- Multi-Select Functionality

  Headstart Oracle Designer and Headstart for Oracle Applications contain features to generate Multi-Select Functionality. Multi-Select functionality was generated for each module component with the template/library object QMSSO$MSEL_BLOCK or AQMSO$MSEL_BLOCK dependent on the Headstart flavor you used. You may also have created a button to perform some action on the selected records. JDG will process these Module Components just as any other Module Component. For more information on multi-select refer to chapter 3 '*JHeadstart Application Generator*' section 'Creating shuttle lay outs'.

  For Multi Select LOV created using Headstart, the JDG will set the multi select property into a Lookup object to Yes.

Attention: A check to identify Headstart typical multi-select blocks is included in the script CheckMCOs.sql.

- Wizard

  If you have created wizards using Headstart you will have one MCO for each wizard page. The JDG will handle these MCO's just as any other MCO, so you will get a VO for each MCO, and a group definition in the Application Structure File.
  You must consider the following:

  - Wizard Page instructions
    With Headstart there were two alternatives to achieve this, but both options used unbound items, so therefore this will not be processed by the JDG. You can include a <text> element to include any text you want to display, or another element including text. View the uix element reference to determine the text element you want to use.

  - Images used for the wizard
    These are created as unbound items, and will therefore not be handled by the JDG. If you do want an image on each page, you must include these in the uix definition on each page using the <image> element.
    Using Headstart you recorded the image you wanted to display between <IM></IM> tags in the hint text, or as a forms parameter.

    For example, lets say you have one image item with the following in the hint text <IM>first_image.gif</IM>. Include this in the appropriate location, typically just after the spacer definition, but before the content with the field definitions, in your generated uix file as follows:

```
(...)
<contents xmlns="http://xmlns.oracle.com/uix/ui">
<spacer height="14"/>
<image source="/app/images/first_image.gif"
       shortDesc="some description"
       hAlign="left"
```

```
        height="100"
        width="100"/>
<labeledFieldLayout columns="" width="10%">
<contents>

(...)
```

- Wizard Buttons
  These are created copying the reusable module component QMS_WIZ_BUTTONS. This MCO will be ignored by the JDG. You will therefore need to add your own buttons to the pages to allow for the proper navigation from page to page

- Whether or not the table is split over more than one page (QMSSO$WIZARD_SPLIT_BLOCK is set for the MCO), and thereby that the content should be content should be committed as part of the same record. In a JHeadstart generated application, you will need to either commit or revert the changes before navigating to another page.

Attention: A check to identify Headstart typical wizards is included in the script CheckMCOs.sql.

- Combination Block
  A combination block is built up of a main MCO and a sub-MCO. The JDG handles typical Headstart for Apps combination blocks as follows:
  - All items in all subcomponents are folded into a single ViewObject and a single group is created in the Application Structure File.
  - The layout style of the group is set to table-form as there is one component with multi-record layout and one with single record layout
  - For all the items in the single record component, the property DISPLAY_IN_TABLE is set to false

- Lov form
  An LOV form is handled just as another form, so therefore it will not be seen as such.

Attention: A check to identify LOV forms is included in the script CheckModules.sql.

- Modal Windows

  Modal windows are recognized in Oracle Designer by the window property Template/Library Object AQMSO$WINDOW_DIALOG (HS4Apps), or starting with QMSSO$MODAL_
  This window should probably pop-up from another non-modal window.
  The JDG does not distinguish between modal and non-modal windows, so if your MCO is based on bound items, then a group in the Application Structure File is generated and a VO reflecting the bound items. If it is only based on unbound items it will be ignored.

  How to handle the modal window in your new application is dependent of the usage. You may consider creating a button in the appropriate window, and open the window by performing using the openWindow function and use the

parentWindow parameter to indicate to which the new window will be modal (see the UIX Developer's Guide for more information about this).

Attention: A check to identify modal windows is included in the script CheckModules.sql.

- Descriptive Flexfields (hs4apps only)

  JDG handles the descriptive flexfield as if it is a (can be bound and unbound) normal item. Developing self-service applications, you will use the OADescriptiveFlexBean to implement your Descriptive Flexfields. See the Developer's Guide for the Self-Service Framework (Oracle Applications) for more details on how to do this.

Attention: A check to identify descriptive flexfield columns is included in the script CheckMCOs.sql.

- Currency format (hs4apps only)

  Any hs4apps specific currency format specified for a column is ignored by the JDG.

Attention: A check to identify columns with specified currency format is included in the script CheckMCOs.sql.

Old Headstart Designer and Headstart for Apps specific constructions (pre Oracle Designer 6i)

If you have migrated to Oracle Designer 6i and have used earlier versions of Oracle Headstart, or if you have only performed a partial upgrade of Oracle Headstart, then you might have forms definitions with old Headstart constructions.
Some of these constructions are ignored by the JDG. Therefore, you make considerations if you have used any of the following:

- Use of QMS_CTRL_BLOCK or AQM_CTRL_BLOCK views

  This was a mechanism used to generate control blocks, as there was no means to create unbound items. JDG ignores any objects that are based on these views. The situation is therefore identical with the situation described in the bullet 'Control block generation' when having control blocks based on bound items.

Attention: A check to identify 'old method' control blocks is included in the script CheckMCOs.sql.

- Row LOV's created from module components with QR_<result_block> naming convention (HS4Apps only)

  Unless you have used the Headstart Upgrade utility to upgrade your Row LOV definitions (or done it manually), you will - after having upgraded to Oracle Designer 6i- have an LOV component with the following naming convention: 'CGFK$QR_<result_block>_<result_item>', and a module component with the naming convention QR_<result block>.

  If you will not use the repository to generate forms in the future, then remove both the module component and the LOV to prevent that JDG generates ADF VO's and a group definition in the Application Structure file. You may consider

to generate a select-form layout to achieve the most similar functionality in your J2EE application.

On the other hand, if you will generate forms on the same module definition, then you must first upgrade your module following the Migration Guide with Headstart for Oracle Apps 11i. Thereafter, see the *JHeadstart Designer Generator Reference* how JDG transforms native Oracle Designer Row LOV's.

Attention: A check to identify 'old method' Row LOV's is included in the script CheckLOVs sql is included in the script CheckMCOs.sql.

- Query Find windows

  Query find windows were generated based on the QMS/AQM_CTRL_BLOCK view object. See the bullet 'Use of QMS_CTRL_BLOCK or AQM_CTRL_BLOCK views' above how JDG handles these objects

- Multi record alternate regions (HS4Apps only)

  Multi record alternate regions were built using the AQM_CTRL_BLOCK to build the region poplist and a module component in display format spread table. If you do not upgrade these forms following the steps in the Migration Guide with Headstart for Oracle Apps 11i, then the JDG ignores the MCO that is based on the AQM_CTRL_BLOCK view. As a result the JDG will generate your remaining Module Component as any other multi-record module component.

Attention: A check to identify 'old method' multi record alternate regions is included in the script CheckMCOs.sql.

- Combination Blocks (HS4Apps only)

  The old method for building a combination block consisted of two MCO's where one was named SUMMARY_<name> and the other DETAIL_<name>. The JDG will generate a group and a ADF VO for both module components. However, they will not co-operate as a combination block did. The most similar behavior to a combination block can be achieved as follows:

  - Before running the JDG, remove the DETAIL_<name> module component
  - After having run the JDG, set the layout style for the generated group to: table-form
  - For the generated View Object, set the DISPLAY_TABLE to true for all the attributes that were displayed in the summary window, and to false for all the other attributes

  If you should keep the form module for future forms generation, you should upgrade your module to use the sub-component mechanism for Combination Blocks as described in the Migration Guide with Headstart for Oracle Apps 11i.

Attention: A check to identify 'old method' combination blocks is included in the script CheckMCOs.sql.

Generate Applications using the JHeadstart Designer Generator

When you have finished preparing your application in Oracle Designer, then you can prepare your project in JDeveloper and use the JHeadstart Designer Generator (JDG) to transform your Oracle Designer application to a JHeadstart application in JDeveloper.

Prepare your projects in JDeveloper

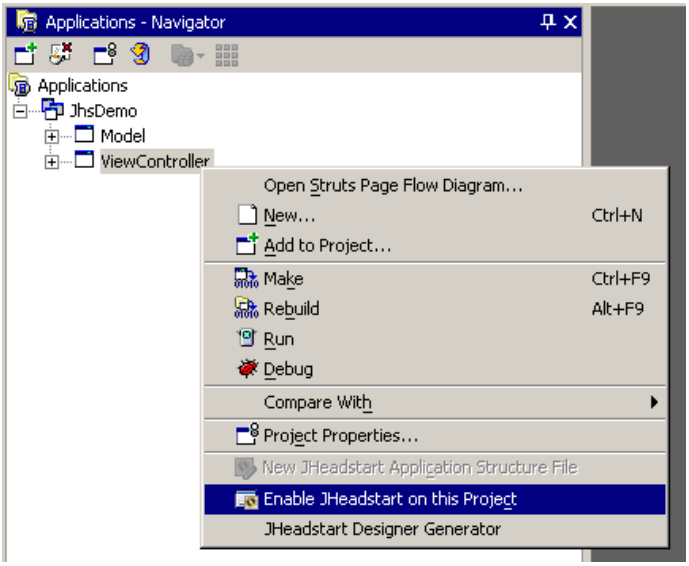
To use the JHeadstart Designer Generator you must first create and setup your Application Workspace with appropriate projects in JDeveloper. You should start the JDG from a model project. This is the project into which you want the JDG to generate the ADF objects. The Application Structure and Domain file should be generated into a specified ViewController project.

See Chapter 2 '*Getting Started*' section 'Setting Up the JDeveloper Project Environment' for information about how to set up your Application Workspace with Model and View Controller projects.

Enabling JHeadstart in the ViewController project

Before you run the JHeadstart Designer Generator, you should enable JHeadstart for the ViewController project. The JHeadstart Designer Generator should be started from the Model project, but the ViewController project must be JHeadstart enabled as the generator also creates files for the ViewController project.

Locate your cursor on the ViewController project, right-mouse click, and select Enable JHeadstart on this Project.



Follow the steps in the wizard.

Creating your connections

When using the JHeadstart Designer Generator you will need two connections:

- A Designer Workarea Connection with a database user and a specified workarea including the containers and objects you want to transform.
- A runtime connection that has access to all the database objects required to run the final application.

You can either create these connections beforehand, or create then while you run the JHeadstart Designer Generator wizard.

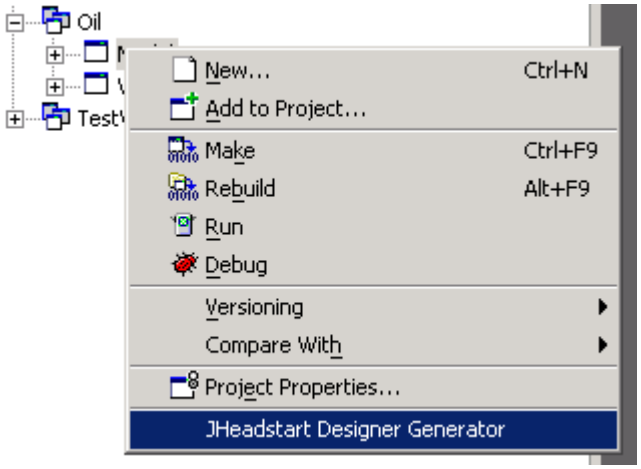
Using the JDG Wizard

When you want to transform your Oracle Designer application into a JHeadstart application you do not have to transform everything in one go. You can choose to start off with a limited set of modules and tables to test the result, and then gradually take new modules and tables until the whole application has been transformed. This enables you to gradually get used to JHeadstart, to verify the JDG result, and make adjustments along the way to maximize the benefits of the JHeadstart Designer Generator.

When you transform a module, you can also easily repeat that transformation at a later point of time. The JDG will then update the previously processed modules according to the changes you made in Oracle Designer.

You should start the JHeadstart Designer Generator from a model project. To start the JHeadstart Designer Generator select your model project in JDeveloper. You can start the generator in two ways:

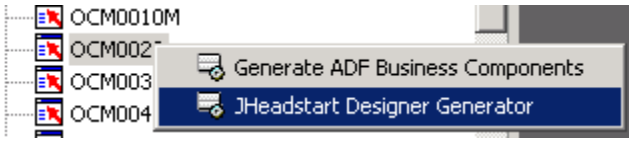
- Use right-mouse click and select JHeadstart Designer Generator. This option will only be available for model projects and projects created with the JHeadstart Project Wizard.
- Use right-mouse click, select New (or from the Menu, select File -> New), go to the Business Components node below the Business Tier. Select JHeadstart Designer Generator.



Note that if for some reason the project has become “instable”, for example through an error in a previous run, it might be that only the second option above is available to start the JDG.

If you want to regenerate a previously generated module, then you must select 'JHeadstart Designer Generator – Remigration' which only will be available when at least one module has been generated before.

As described in the Introduction of this chapter, the JHeadstart Designer Generator extends the ADF Business Components from Oracle Designer Generator delivered with JDeveloper. This generator will in various situations be available to you when the JHeadstart Designer Generator is available.
For example, when you want to start the JDG from the New Gallery (choosing Business Components in the Business Tier), you will see both the "Business Components Generated from Oracle Designer" (the default generator), and, a bit further down, the "JHeadstart Designer Generator".
Also, if you try to start the JDG directly from a workarea connection (instead of from a project), you will see both generators using right-mouse click:



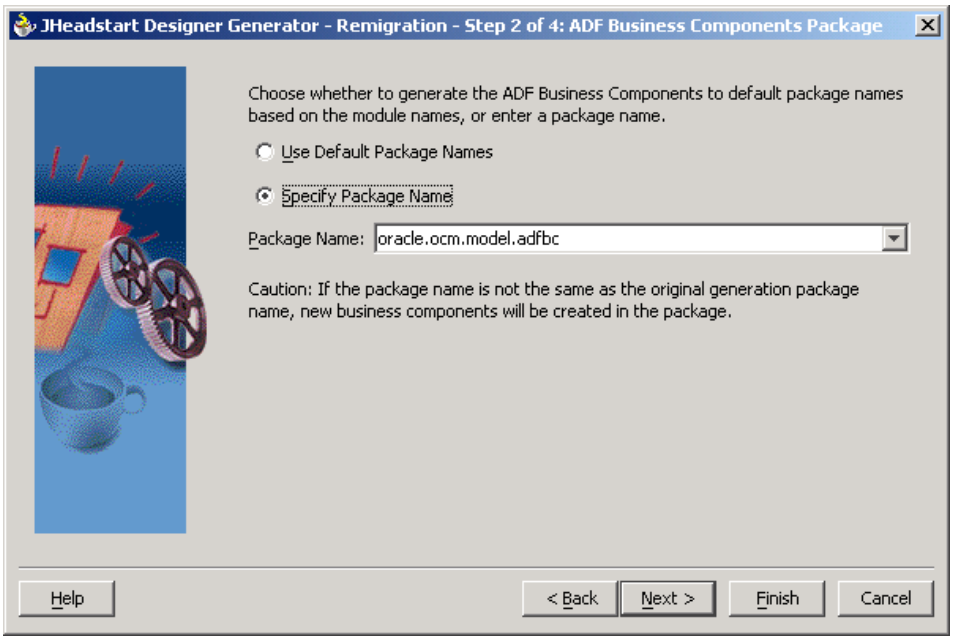
It is recommended that you start the JHeadstart Designer Generator from the Model project node, instead of from the workarea connection (as shown here), as the project context is not given up front, and therefore a number of default settings may not be correct.

Process your Oracle Designer Modules

You will typically begin by processing some of your easiest Oracle Designer Modules. You should perform the steps below to process Oracle Designer Modules. Before you start processing your modules, you should have done the proper preparations in Oracle Designer as described in the section 'Prepare in Oracle Designer for generation' earlier in this chapter.

Attention: View the viewlet JDGModuleTransformation for a demonstration on how to process a module from Oracle Designer using the JDG.

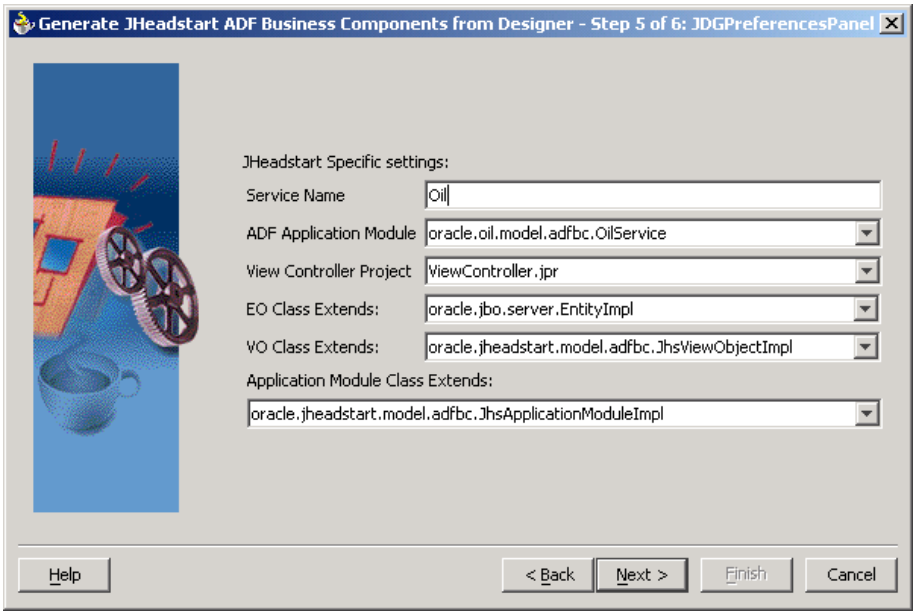
1. Start the JHeadstart Generator, and press next until the following window is displayed.



It is important that you specify a package name, otherwise you will get one package for each module you select, and the JHeadstart Application Generator is not built to handle that. You will get an error if you continue without specifying a package. Do not ignore this. If you are using JHeadstart Designer Generator multiple times in a project, to select a sub section of Designer modules every time, make sure that you always choose "Specify Package Name" and without fail, specify the same package name every time.

Do not use the option "Use Default Package Names" as JHeadstart Application Generator is not fully prepared to handle default packages.

2. Press the Next button in the JHeadstart Designer Generator Wizard to enter the JDG parameters.



Enter the name of the service. You would normally choose the name of your application.

Enter the name of your ADF application module. You would normally choose a name using the following naming convention: packagename.YourappService.

Choose the ViewController project. This is the project in which the Application Structure File and the Domain Definition File is generated. The ADF Business Components are generated into the project you started the JDG from. The JDG sees this as the model project.

Amongst the recommendations the *Business Rules White Paper*, you are advised to create your own EntityImpl super class that in turn extends oracle.jbo.server.EntityImpl. This is handy for example to handle recording audit columns for your application, as well as providing an invalidateMe() method.
If you have created your own custom Base Class for Entity Objects, then enter your own base class for the EO Class Extends. The JDG will then ensure that every Entity Object created or updated by the JDG extend your base class instead.

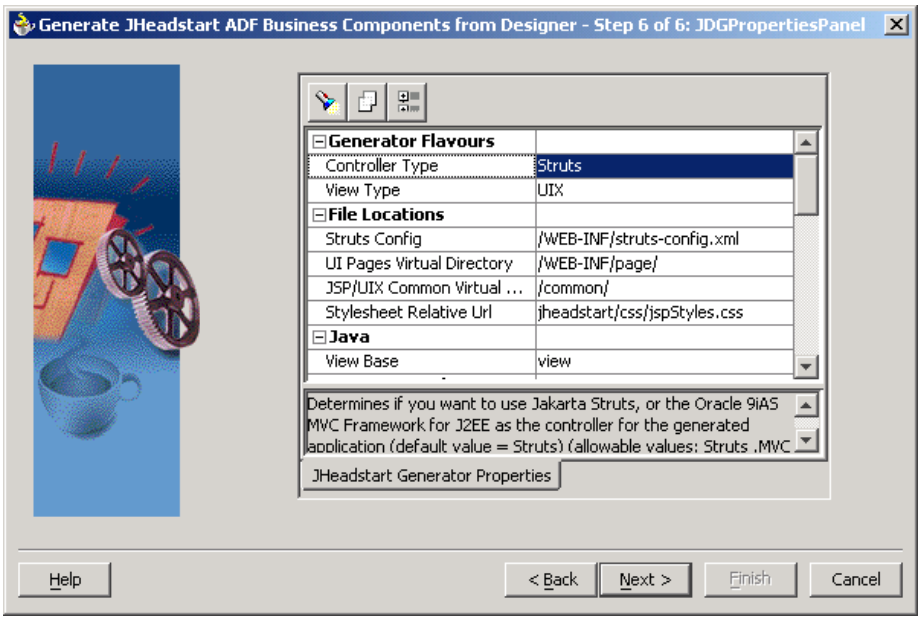
You can also make your own View Object Base Class that can be used to extend all your View Objects. If you have created your own custom Base Class for View Objects, then enter your own base class at VO Class Extends. The JDG will then ensure that every View Object created or updated by the JDG extend your base class instead.

If you are using CDM RuleFrame, then you must make certain that your Application Module extends the oracle.jheadstart.model.adfbc.RuleFrameApplicationModuleImpl instead of the default oracle.jheadstart.model.adfbc.JhsApplicationModuleImpl. This will ensure that your application module uses the CDM RuleFrame transaction classes to open and close transactions and to handle any errors placed on the CDM RuleFrame message stack.
Select the oracle.jheadstart.model.adfbc.RuleFrameApplicationModuleImpl in the Application Module Class Extends combo box to do this.

If you have your own application module implementation, you just enter the full class name in the combo box.

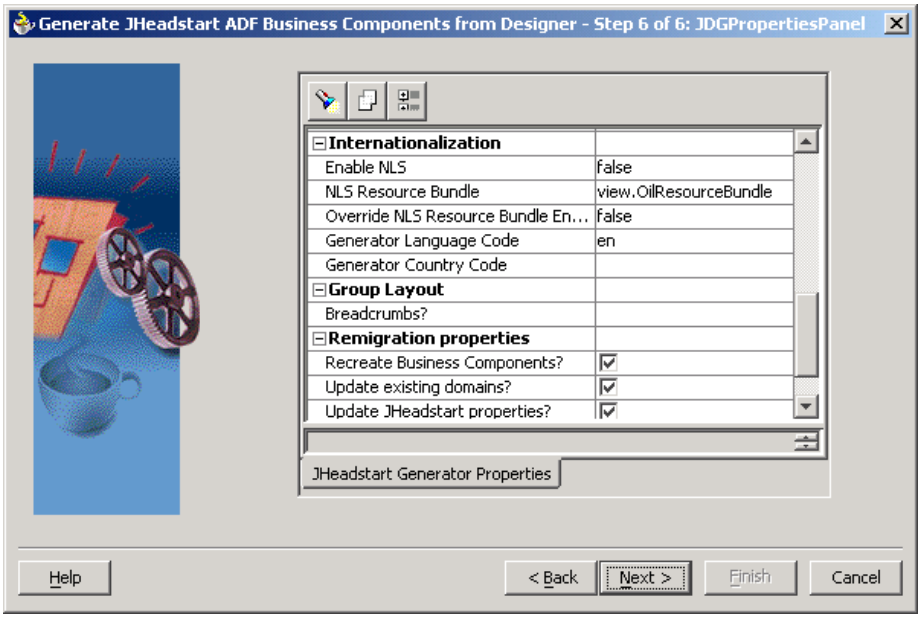
Note that any extensions will only work if the classes to extend are available in the project you run the JDG from.

3. Press the Next button in the JHeadstart Designer Generator Wizard to enter various properties. These properties will be mainly be transformed into the service level of your Application Structure File.



Below the properties, you see a small area with the help text. If you click on a property, the help text for that property appears in the window. This area may seem unnecessary small. You can increase or decrease the size of this area as you desire, just by placing the mouse cursor on the line above the text and move the line up or down.

Further down you see some Remigration properties:



These are used only when you are re-running the JDG on objects that have already been generated. The recreation of Business Components is checked by default. If you uncheck this property the Business Components will not be recreated, but if you leave the Update JHeadstart properties checked all the JHeadstart specific properties on the objects will be updated unless you have set the generatorProtected property for the object to true.

You can use these settings to ensure that you don't overwrite post-generation changes you have made.

4. When you press the Next button in the JHeadstart Designer Generator Wizard you can view the summary of your selections. If that looks correct, then press Finish to start the JHeadstart Designer Generator.

Checking the generated result

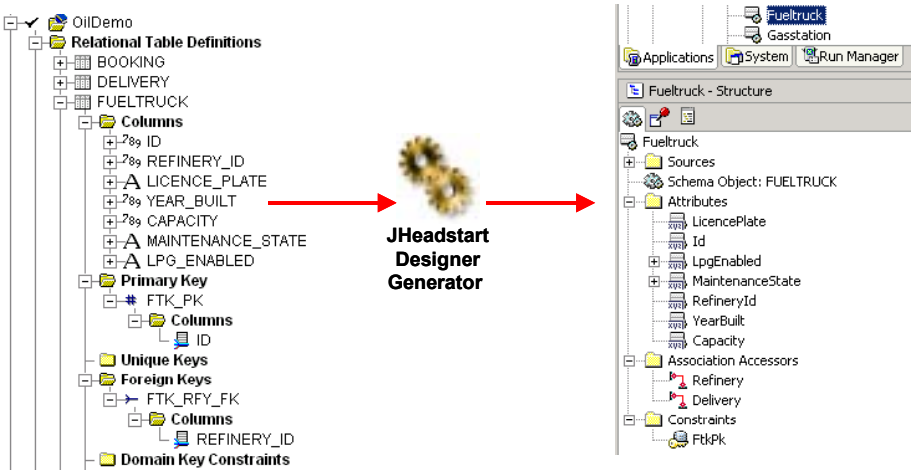
When you have used the JHeadstart Designer Generator to process the various Oracle Designer objects you can view the actual result generated result in your project in JDeveloper.

Checking processed modules

For all the processed modules the following has been generated:

- Entity Objects (EO) and Associations

  A module is based on one or more module components (MCO) with one or more table usages. For each table usage in a module, an ADF Entity Object is created unless it already existed. If it did exist, and you specified to recreate Business Components, then the Entity Object is updated when there are changes.

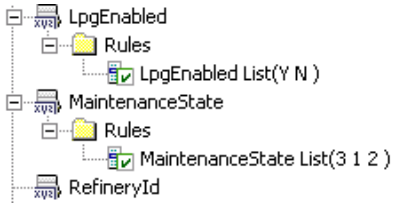


  For each column, an attribute has been created.

  For each foreign key, an association has been created. In the example above, you can see no foreign keys to DELIVERY in Oracle Designer. The generated associations are foreign keys from another object to FUELTRUCK. In Oracle Designer there is one foreign key from DELIVERY to FUELTRUCK.

  For each primary key a Key constraint is generated. In the example above you can see that a key has been generated for the primary key as defined in Oracle Designer.

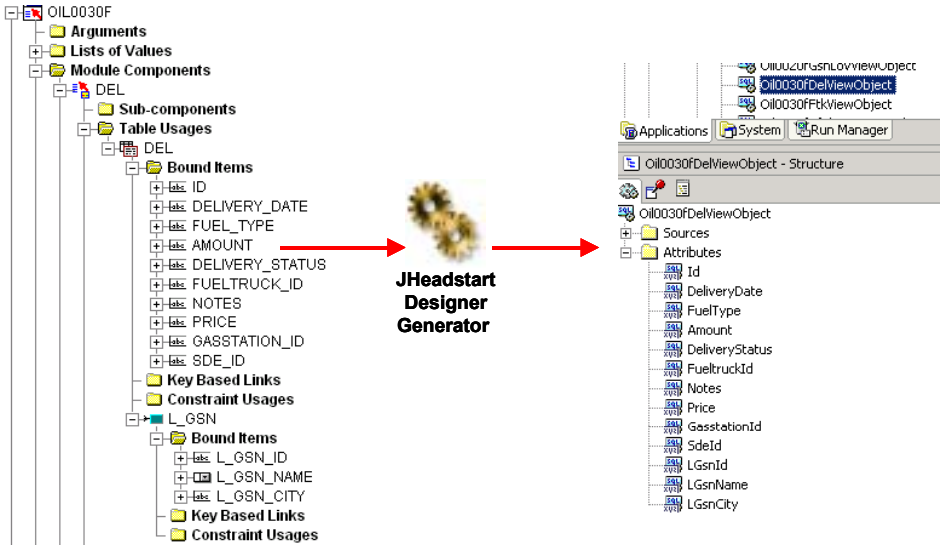
  You can se a + ahead of two attributes, the LpgEnabled and MaintenanceState. This is because a domain was linked to the corresponding table columns in Oracle Designer. In that situation a validator rule is created for the attribute:



To see how each Designer property of columns, keys etc are translated to ADF objects see the JDG Reference.

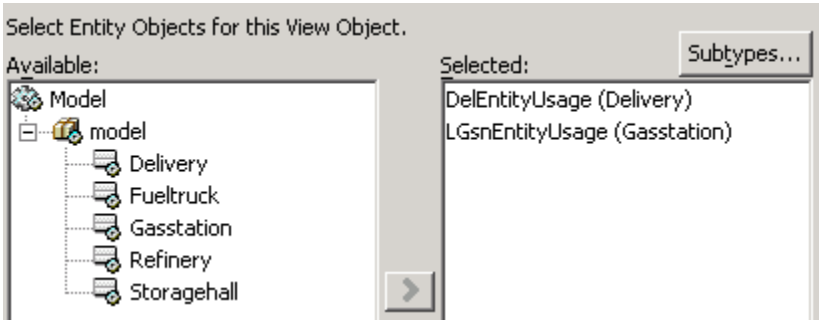
- View Objects (VO)

  A module is based on one or module components. For each module component a view object is created that includes Entity Objects for the base table and lookup table usages in the Module Component.



For each Bound Item an Attribute has been created in the VO. This includes the lookup bound items. The order of the attributes is the same as the display order of the MCO items if no item groups are used in the MCO. If item groups are used, then the items within the item group are grouped together as follows: When the first item in an item group is found, then the other items in the item group are processed immediately, before any other items not in the item group even though these may have lower display sequences than some of the items in the item group.

If you edit the view object, you will see that the lookup tables have been included in the view object using the generated associations:



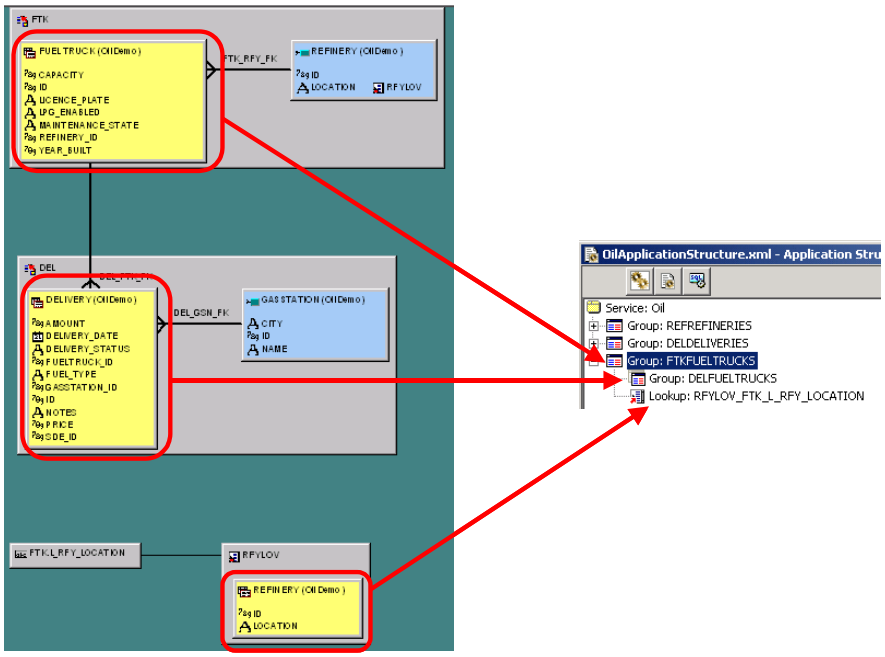
For more detail on how MCO and item properties are processed by the JDG see the JDG Reference.

- Application Structure File

  For each module component in the module a group definition has been created in the Application Structure file.
  Module components that are details of a master module component are created as nested groups to the master group.
  Master-Detail module components located on the same window are generated to be located on the same page.

  At this moment, since the JAG does not support more than two levels Master-Detail, the JDG will create as a Root (Master) Group any MCO that has a Child MCO. MCO's that are CHILD to some other MCO are also processed into a nested Group; that means that one MCO can lead to two Groups: once as a nested group and once as a master group.

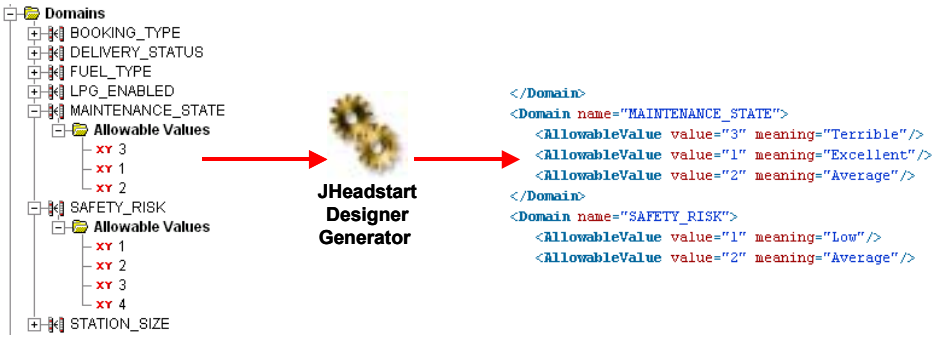
  All LOV inclusions (link between an Item in an MCO and a List of Values object) are created as lookups in the group definition.



For more detail on how MCO and item properties are processed by the JDG see the JDG Reference.

- Domain Definition File

  For each static and dynamic domain included in the generation an entry is created in the Domain Definition File.



The value and meaning are taken from the domain's allowable values.

Some hints on how to improve the performance of the final application

The performance of the final application can be improved if you remove any Entity Objects that are based on a non-updateable database view, and change the View Object that were based on these Entity Objects to be read-only views. This will save on cashing and memory usage in general.

This can be done as follows:

1. Select the View Object, right mouse-click, and then Edit <View Object>.
2. Navigate first to the Query node and activate the Query Expert Mode by checking the Expert Mode checkbox.
3. Thereafter, navigate to the Entity Object tab and remove the link to the Entity Object. If you now move back to the Query Tab, you will see that the query is still there, and that the Expert Mode checkbox is deactivated. Press OK.
4. Repeat the steps above for all View Objects based on the Entity Object
5. Remove the Entity Object.

Further Steps

Now you can use the JHeadstart Application Generator to generate your JHeadstart Application, and where necessary Customize your generated application. See Chapter 3 *'JHeadstart Application Generator'* how to use this generator.

CHAPTER

5

JHeadstart Extensions to ADF Runtime

This chapter discusses the extensions made by JHeadstart to the ADF Runtime. The first two sections provide an architectural overview of how ADF and JHeadstart submit and handle HTTP requests. It explains the differences between using UIX or JSP as the view technology, and how JHeadstart minimizes these differences by smart architectural choices.

The remainder of this chapter discusses how all the features as described in the chapter "JHeadstart Application Generator" are implemented with highly generic code at runtime. These features are divided into the following sections:

- User Interface Widgets
- Page Design
- Query Behaviors
- Transactional Behaviors
- Message Handling and Internationalization

With the information in this chapter, you should be able to find your way in the JHeadstart runtime code, allowing you to extend or override default JHeadstart runtime behavior.

Reference: When you see a reference like this, you are referred to JHeadstart Javadoc. The easiest way to find this Javadoc is by going to your JHeadstart (ViewController) project in JDeveloper, and choosing the menu option Navigate – Go to Java Class (or use the shortcut keys Ctrl+Minus). You can use the same feature to go to the Java source code instead of the Javadoc, which can be useful for setting debugging breakpoints (or simply for knowing exactly what is being done).

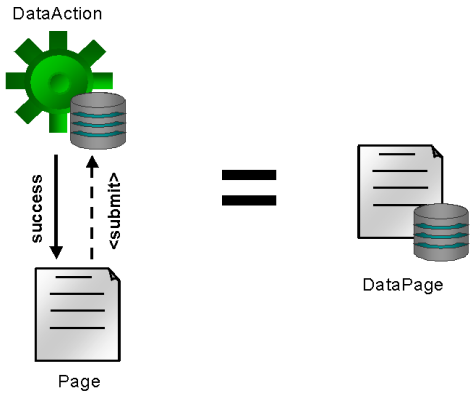
HTTP Request Handling

Using ADF in a Model-View-Controller architecture (also known as Model II), the Struts Controller handles requests from both UIX and JSP pages. Struts matches the request URI (Unified Request Identifier) with an action mapping as defined in the struts-config, and executes the action class associated with this mapping. For example, the URI '`/myapplication/StartMyAction.do`' is matched to the action mapping with path '`/StartMyAction`' in the `struts-config.xml` file.

Struts executes the action by calling the `execute()` method of the action class. For this to work the action class must be a subclass of `org.apache.struts.action.Action`. The `execute()` method returns a forward as defined in the struts-config, which determines which page must be displayed after the action has executed, or, when the forward refers to another action mapping, which action needs to be executed next.

ADF Data Action and Data Forward Action

Using ADF the action class executed will be (a subclass of) `oracle.adf.controller.struts.actions.DataAction` or `oracle.adf.controller.struts.actions.DataForwardAction`. The latter will be used when creating a *data page* as opposed to a *data action*. The difference between these two is largely cosmetic: In the Struts Page Flow Diagrammer a data action that forwards to a page results in two icons: the action and the page. A data page is rendered as a single icon.



The only functional difference between the two classes is the way the default forward of the action is determined. In case of a data page, ADF will evaluate the "parameter" property of the action mapping, in case of a data action ADF will return the "success" forward of the action mapping.

Here is an example of a data page:

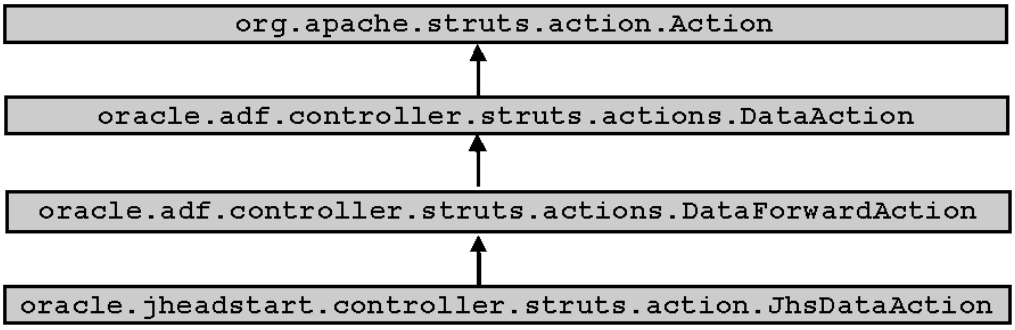
```
<action
    path="/depts"
    className="oracle.adf.controller.struts
               .actions.DataActionMapping"
    type="oracle.adf.controller.struts
               .actions.DataForwardAction"
    name="DataForm"
    parameter="/depts.jsp"
>
```

```
    <set-property property="modelReference"
                  value="deptsUIModel"/>
  </action>
```

And here is the same example using a data action:

```
  <action
      path="/depts"
      className="oracle.adf.controller.struts
                .actions.DataActionMapping"
      type="oracle.adf.controller.struts
                .actions.DataAction"
      name="DataForm"
  >
    <set-property property="modelReference"
                  value="deptsUIModel"/>
    <forward name="success" path="/depts.jsp">
  </action>
```

Many of the JHeadstart runtime behaviors have been implemented in `oracle.jheadstart.controller.struts.action.JhsDataAction`. This class is a subclass of `DataForwardAction`. So, the overall action inheritance tree looks like this:



Although an action mapping that uses `JhsDataAction` will render as a data page on the Struts Page Flow Diagrammer, it can be used as a data action as well. It combines the default forwarding behavior of `DataAction` and `DataForwardAction`: if the '`parameter`' property is not set, or set to "`unknown`", it will evaluate the "`success`" forward.

As explained above, Struts will call the `execute()` method on the action class. To understand how you must extend or override ADF runtime behavior, it is useful to take a look at the `execute()` method of the `DataAction` class:

```
public final ActionForward execute(ActionMapping mapping,
                                    ActionForm form,
                                    HttpServletRequest  request,
                                    HttpServletResponse response)


      throws Exception
{
  // Configure the context for this DataAction
  final DataActionContext actionContext = new DataActionContext();
  actionContext.initialize(this, mapping, form,
                           request, response);
  // Start the execution of the lifecycle
  handleLifecycle(actionContext);
  return actionContext.getActionForward();
}
```

For seasoned Struts users, defining the `execute()` method as `final` will look like a deadly sin. However, when using ADF, it makes perfect sense. By making the method

final, ADF ensures that initialization code related to the data bindings is always executed before actual execution of the action logic. Furthermore, it creates a handy `DataActionContext` object that provides access to all objects passed into the execute() method, as well as to other data binding related objects you will need access to almost every time you will write your own DataAction subclass:

- The Binding Context that provides access to any Binding Container (also known as UI Model) of your application.
- The Binding Container of the current action mapping, as specified in the '`modelReference`' property of the Struts action mapping.

So, when you want to write your own DataAction subclass to extend or override default behavior, you can overwrite method `handleLifecycle()` instead of the `execute()` method. However, in many situations, you will not need to override the `handleLifecycle()` method either. The `handleLifecycle()` method can be seen as the "main routine" that calls a series of "sub" methods in a set order. These methods will do the actual work, so often it might be sufficient to override one of these "sub" methods. To understand when to overwrite what method, it is important to first understand the phases that ADF distinguishes in handling the page lifecycle.

ADF Page Lifecycle

ADF distinguishes the following phases in handling the page lifecycle (between brackets the name of the associated method in the `DataAction` class):

- Build Event List (`buildEventList()`)
- Prepare Model (`prepareModel()`)
- Update Model (`processUpdateModel()`)
- Validate Model Changes (`validateModelUpdate()`)
- Process Page Events (`processComponentEvents()`)
- Invoke Data Action Method (`invokeCustomMethod()`)
- Report Errors (`reportErrors()`)
- Dispatch to Correct Page / Action (`findForward()`)

Reference: See the Javadoc of the `handleLifecycle()` method of `oracle.adf.controller.lifecycle.PageLifecycle`.

These phases are the same for UIX and JSP. However, the implementation of some of these phases is different dependent on whether UIX or JSP is used. For this reason, the actual logic of each phase is coded into View-specific lifecycle classes: `oracle.adf.controller.struts.actions.StrutsJSPPageLifecycle` for JSP and `oracle.adf.controller.struts.actions.StrutsUIXLifecycle` for UIX. Note that when you want to debug this code, it is useful to know that both classes are part of the same inheritance tree, so some methods might be implemented in a common superclass. See also section 'JHeadstart PageLifecycleFactory'.

The DataAction method for each phase delegates to the corresponding method in the lifecycle class, for example:

```
protected void processUpdateModel(DataActionContext
                                  actionContext)
{
  Lifecycle cycle = getPageLifecycle(actionContext);
  cycle.processUpdateModel(actionContext);
}
```

The reason all the lifececycle methods are still included in the `DataAction` is to facilitate customization. When extending or overwriting default behavior, you can always override the method in the DataAction, you do not have to worry about the lifecycle classes because normally you will only use one view technology in your application. For JHeadstart this is slightly different. Since the JHeadstart runtime code must support both JSP and UIX, JHeadstart includes two page lifecycle subclasses that implement view-specific extensions.

In the next subsections, we will explain in more detail what happens in each phase of the page lifecycle, and will highlight differences between JSP and UIX where appropriate.

Build Event List

The execution of this phase is the same for JSP and UIX. It builds the list of events with their possible associated action binding from the request parameters. By doing this once and storing the result in the lifecycle context the other lifecycle phases don't have to walk through the request parameter list again to get the events.

Prepare Model

The execution of this phase is the same for JSP and UIX. ADF will loop over all iterator bindings in the current UI Model (binding container), and will query the iterator:

- if it has not been queried before, and
- the iterator has not been set in *find mode*, and
- the fetch size of the underlying ViewObject (when using ADF Business Components) is not set to zero.

Note that when using ADF Business Components, it will actually query the underlying ViewObject Usage of the Application Module. So, if you use the same ViewObject Usage in multiple pages, the query will only be executed when you start the first of those pages for the first time.

Since ADF automatically queries all iterator bindings that meet the above conditions, you must apply any query bind parameters that you have set up in your ViewObject query definitions before that happens. In other words: override `prepareModel()` and set up the query bind parameters before you call `super.prepareModel()`. This is exactly what JHeadstart has done to implement the query bind parameter functionality provided by the JHeadstart Application Generator. See section 'Query Bind Parameters' for more information.

Update Model

In the Update Model phase, the changes made in the page are written back to the corresponding bindings in the model layer. The way this is done is different for UIX and JSP.

Using JSP, the model update will be perfomed using the form bean associated with the action mapping. Struts populates the form bean using the request parameters, and the ADF `StrutsPageLifecycle` class will write the changes to the model.

Using UIX, the `StrutsUIXLifecycle` class calls out to the UIX Controller to perform the actual update. The UIX Controller uses the `InitModelListener` class as specified in the web.xml to do the model update:

```
<init-param>
    <param-name>oracle.cabo.servlet.UIXRequestListeners</param-name>
    <param-value>oracle.cabo.adf.rt.InitModelListener</param-value>
</init-param>
```

So, to customize or extend the behavior of the `InitModelListener` class, you can specify your own `InitModelListener` class.

That is exactly what JHeadstart has done. In order to allow certain customizations to the default UIX behavior, JHeadstart includes a `JhsInitModelListener`. Two of those customizations, validating model updates and multi-part request handling, are discussed in the next few pages.

Reference: See the Javadoc of `JhsInitModelListener`.

Validate Model Changes

In this phase, the model updates are validated. This phase is different for UIX and JSP.

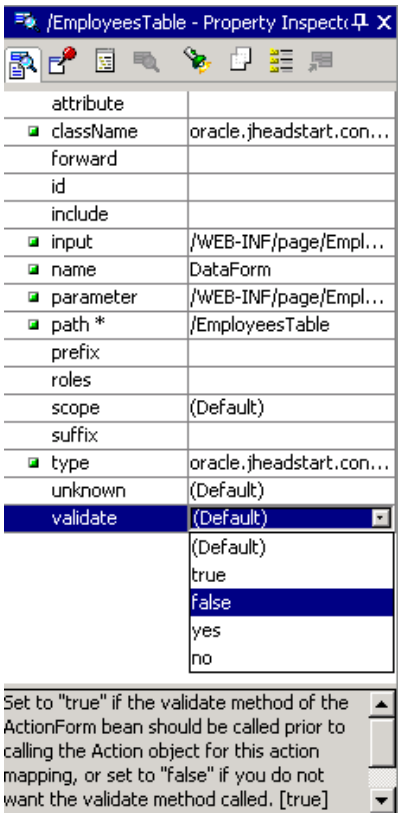
For JSP in combination with ADF Business Components this phase results in a call to the `validateEntity()` method of each Entity Object that has been changed during the previous phase. Typical model exceptions reported are required attributes without a value.

Extending or switching off model validation when using JSP is easy: simply override method `validateModelUpdates()` in your DataAction subclass.

Using UIX, model validation cannot be extended or switched off easily (bug 4028488). This is due to the fact that model validation is performed implicitly as part of the model update by the private method `_doModelUpdate()` in the InitModelListener class.

JHeadstart comes to the rescue here: the `JhsInitModelListener` class, which is based on, but not a subclass of `InitModelListener` does not perform model validation in the _doModelUpdate() method. Instead model validation is handled by the JhsDataAction in the same way as it is done for JSP. This has the additional advantage that model validation only takes place after all updates, both single-row *and* multi-row updates, are done.

Furthermore, JHeadstart provides declarative support for a common customization: switching off model validation. This might be helpful when building a wizard-style user interface where required attributes are spread over multiple wizard pages. JHeadstart uses the Struts action mapping property '`validate`' for this purpose. If this property is set to "`false`", model validation will not be performed.



For JSP, this has been implemented in `JhsDataAction.validateModelUpdates()`. For UIX, this has been implemented in the method `JhsInitModelListener.validateInputValues()`.

Reference: See the Javadoc of `JhsDataAction.validateModelUpdates()`, and `JhsInitModelListener.validateInputValues()`.

Process Page Events

When using JSP's, page events are always handled by the DataAction and associated lifecycle class. When using UIX, *by default* events are handled by event handlers defined inside the UIX page. This difference has some far-reaching implications, as we will explain in the next sub-sections. Fortunately, it is possible to change the UIX pages to have events handled in the same way as JSP. But first, we will explain the default behavior for each view type.

JSP Event Handling

When you drag and drop a standard operation or custom method as a button or hyperlink onto your page, you are adding an event to your page. When the user clicks the button or hyperlink, the event is sent as a request parameter. The event parameter can take two forms, depending on whether you use a button or hyperlink to submit the page:

When dragging the Commit operation as a button onto your JSP page, it will look like this in the JSP source:

```
<input type="submit" name="event_Commit"  value="Save"/>
```

You could also use a hyperlink to submit the Commit event:

```
<a href="deptTable.do?event=Commit">Commit</a>
```

When the button or hyperlink is clicked, by default the `processComponentEvents()` method will execute the action binding with the same name as the event. If such a binding does not exist, nothing will happen. You can change this default behavior by including an `on<event_name>()` method, for example `onCommit()`, in your DataAction subclass. You can add add your own code to this method, and to execute the binding associated with the event, you can use the following statement:

```
actionContext.getEventActionBinding.doIt();
```

Note that you cannot call super, because the ADF Data action does not include onEvent methods.

This is a simple yet powerful technique that allows you to extend or overwrite the default behavior of standard operations you have dragged onto your page. But you can also use an onEvent method for custom events. If you include an event named `foo` in your page, the `onFoo()` method in your data action will fire. You can put any logic you like in this method. Your binding container (UI Model) does not need to include a binding named foo.

UIX Default Event Handling

When using drag and drop to add a standard action or custom method to a UIX page, by default the associated event will be handled by a page event handler that is also added to the page through the drag and drop action. When you drag the Commit operation to a UIX page, the following code snippets are added to the UIX page source:

```
<submitButton text="Commit" model="${bindings.Commit}"
 id="Commit1" event="action"/>

<handlers>
  <event name="action" source="Commit1">
    <invoke method="doIt"
      javaType="oracle.jbo.uicli.binding.JUCtrlActionBinding"
      instance="${bindings.Commit}"/>
  </event>
</handlers>
```

As you can see, the UIX event handler directly calls the `doIt()` method on the action binding, bypassing Struts as the Controller. Since the event is named `"action"`, an `onCommit()` method in the data action would not fire either (bug 4175221). Furthermore, the powerful event-based forwarding as explained in the section 'Dispatch to Correct Page / Action' cannot be used.

Fortunately, it is quite easy to use the same event handling for UIX, as is used for JSP. All you have to do is change the UIX source that was created using drag-and-drop from the Data Control palette. If you go to the properties of the button associated with the named event, and change the value of the `event` property from `"action"` to `"[EventName]"` (`event="Commit"` in the example), then the Struts data action will handle the named event, just like with JSP.

```
<submitButton text="Commit" model="${bindings.Commit}"
 id="Commit1" event="Commit"/>
```

You can then remove the event handler for "action" with source "Commit1" at the bottom of the UIX source, because it won't be used anymore.

Generation: JHeadstart generated pages always use the Struts data action for handling named events, both with generated JSP pages and generated UIX pages. For UIX pages JHeadstart uses the abovementioned technique of setting the event property of the button equal to the event name.

JHeadstart onEvent methods and the eventValue parameter

To illustrate the power of onEvent methods: the JhsDataAction class contains no less than 14 onEvent methods! For example, JHeadstart includes an onDelete() method that will perform an auto-commit (by calling the onCommit() method) to ensure the row will be deleted from the database right away.

One specific problem we had to solve is a page where the same event was needed multiple times. For example, a master-detail page where both the master and detail can be deleted. When using drag and drop to implement this functionality, you get two button definitions like this:

```
<input type="submit" name="event_Delete"  value="Delete"/>

<input type="submit" name="event_Delete1"  value="Delete"/>
```

In the page UI Model, two action bindings are added: Delete and Delete1. The Delete binding is associated with the iterator binding of the master, the Delete1 binding is associated with the iterator binding of the detail.

To get the auto-commit behavior with the delete, two methods need to be added to the data action: onDelete() and onDelete1(). For your own custom subclass, this would be OK, for JHeadstart this was not an option since the JhsDataAction is highly generic and we cannot know in advance how many delete events are fired from a single page.

JHeadstart has solved this problem by using an additional request parameter named eventValue. The event parameter will be named "Delete" for both buttons, the eventValue parameter will contain the name of the actual action binding that must be executed: "Delete" or "Delete1". In the onDelete() method, the eventValue parameter is evaluated and the corresponding action binding is looked up and executed.

In a JHeadstart-generated JSP page the buttons will use a JavaScript function that will look like this:

```
<input type="button" onClick="doEvent('Delete','Delete');"/>

<input type="button" onClick="doEvent('Delete','Delete1');"/>
```

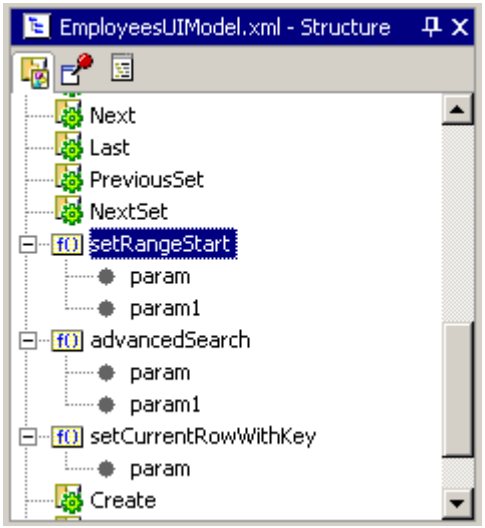
The JavaScript function copies the first argument to a hidden form field named event, and the second argument to a hidden form field named eventValue. See section 'Request Submission - Navigation Events and Data Events' for more information about this JavaScript function.

Passing Action Binding Parameters

If the named event is associated with a custom method action binding that has one or more parameters, you have two ways to specify the values for these parameters:

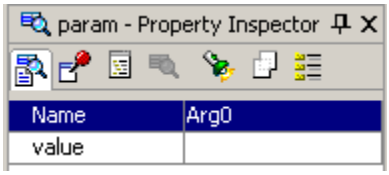
- Declaratively, through the property inspector on each argument that can be selected by expanding the action binding in the UI Model pane of your page.
- Programmatically, by adding an onEvent method to your data action. In this method you first create an `ArrayList` of parameter values, then set this list on the method action binding by calling `setParams()` on the method binding, and finally execute the method by calling `doIt()` on the binding. See `JhsDataAction.onQuickSearch()` for an example of this technique.

Below, we will discuss the declarative way in more detail. First, we explain the standard ADF functionality provided in this area, then we will dicuss how JHeadstart has improved this functionality to meet the JHeadstart needs.



Example UI model with action binding setRangeStart that has two parameters

Action binding parameters have two properties that you can set in the UI model: a `name` and a `value`.



With default ADF, the only ways that you can pass values to those parameters using these `name` and `value` properties are:

- All parameter `name` properties must be a valid request parameter name. If one of the parameter names does not specify a request parameter, then none of them are passed.
- The `value` property must refer to a UI model and a value binding in that UI model, for example: `DeptUIModel.Deptno`. Note that you cannot use an Expression Language statement, like you can when setting up parameter values for a method dropped onto an action (bug 4175157).

JHeadstart has changed this behavior to be in line with the way you can pass parameters to action bindings in Struts action mappings, so that you can use the `value` property to:

- Pass a literal `String` value as parameter

- Pass a request parameter
- Pass an EL expression, which allows you to specify a data binding like `${data.DeptUIModel.Deptno}` or (if `DeptUIModel` is the current binding container) `${bindings.Deptno}`.

Besides the similarity to the way you pass parameters to methods dropped onto a Struts data action, this gives you much more flexibility in passing parameters via the UI model (with default ADF behavior, it is not possible to pass literal values, nor a mix of request parameters and bindings).

This is implemented by overriding the `Lifecycle` method `initializeParameterValuesUsingRequestObject()` in both `JhsStrutsUixLifecycle` and `JhsStrutsJspLifecycle`. For more information about these JHeadstart `Lifecycle` classes, see section JHeadstart PageLifecycleFactory

Reference: See the Javadoc of the `initializeParameterValuesUsingRequestObject()` method of `JhsStrutsUixLifecycle` and `JhsStrutsJspLifecycle`.

Invoke Data Action Method

This phase is simple and the same for UIX and JSP: when you have dragged and dropped a standard operation or custom method to your data action or data page in the struts-config (as opposed to dragging it onto a page), the `invokeCustomMethod()` will execute the method. So, despite the name of the method, it will also execute standard operations dropped onto the action. Note that the method you dropped will be executed *every time* the action is executed.

Report Errors

This phase is different for UIX and JSP.

For JSP, binding container runtime errors are transferred to the Struts `ActionErrors` collection.

For UIX, the InitModelListener class handles this functionality just before rendering a page: binding container runtime errors of type `JboException` are converted to UIX `MessageData`.

Customized Handling of Model Exceptions

The ADF Toy Store demo by Steve Muench customized the default way that a "tree" of bundled ADF exceptions gets translated into Struts `ActionError` objects for display in the JSP page by overriding the `reportErrors()` method.

If multiple exceptions are reported for the same attribute, he simplifies the error reporting by only reporting the first one and ignoring the others. An example of this might be that the user has provided a key value that is a duplicate of an existing value, but also since the attribute set failed due to that reason, a subsequent check for mandatory attribute ALSO raised an error about the attribute's still being null.

Reference: The ADF Toy Store Demo, including an excellent technical whitepaper by Steve Muench explaining how the demo was built, can be found at http://www.oracle.com/technology/products/jdev/collateral/papers/10g/adftoystore.html.

JHeadstart has copied this Toy Store extension to the JHeadstart `ReportingUtils` class and made some further customizations:

- You can configure whether the stack trace of unexpected exceptions is printed. By default it is printed. You can call method `setPrintStackTraceUnexpectedExceptions(false)` to switch this off.
- You can configure which exceptions should be treated as expected exception, for which the stack trace will never be written. By default `ValidationException`, `TooManyObjectsException` and `JhsJboException` are treated as "expected" exceptions.
- You can configure whether the error code (product code and error number) should be displayed through method `setShowErrorCode()`.
- When the underlying exception is a `SQLException` that indicates a database constraint violation, a message with the constraint name as key will be added to the `ActionErrors`, so you provide a user friendly message to the user, by adding this constraint name as key to your message resource bundle.

All these static configuration methods can be called from anywhere you like in your code.

Reference: See the Javadoc of `JhsStrutsJspLifecycle.reportErrors()`, and `ReportingUtils`.

Displaying Struts Action Errors/Messages in UIX

With default ADF UIX, only model exceptions of type `JboException` are shown to the end user. Struts action errors are ignored. A possible workaround could be to write code to throw a new `JboException` for each Struts `ActionError` object, but then still the informational Struts action messages (`ActionMessage` objects) would not be shown. Besides, with JSP we have tags to show Struts action errors and action messages in the page, so it would be nice if UIX could do that too.

JHeadstart solves this by converting Struts `ActionError` / `ActionMessage` and ADF `JboException` objects to a UIX `MessageData` object, so the standard ADF UIX message box will display them to the user:

```
<messageBox model="${data}"/>
```

This is implemented in the `JhsUixStrutsLifecycle` class (for more information about this class see section 'JHeadstart PageLifecycleFactory'), by overriding the ADF lifecycle phase `reportErrors()`. This method has three stacks of messages to convert:

1. Model exceptions that are stored on the binding container (= the runtime representation of the page UI model). These are converted to Struts Action errors using the `ReportingUtils` described in the previous section, so that they can be picked up in the third step.

2. Standard Struts action messages, stored on the request with key `Globals.MESSAGE_KEY`, are converted to UIX MessageData.

3. Standard Struts action errors (including the result of step 1), stored on the request with key `Globals.ERROR_KEY`, are converted to UIX MessageData.

These messages are collected in a single UIX `MessageData` object, which is then stored on the request under the key '`_uix.messages`'.

To ensure that the UIX servlet does not clear this '_uix.messages' entry, the method JhsInitModelListener.renderStarted() is different than its counterpart method in the default UIX InitModelListener class (for more information about JhsInitModelListener see section 'Update Model'). If the '_uix.messages' entry already has a value, it is not cleared.

Reference: See the Javadoc of JhsStrutsUixLifecycle.reportErrors(), and JhsInitModelListener.renderStarted().

Dispatch to Correct Page / Action

ADF supports event-based forwarding, a powerful feature that allows you to do conditional page navigation without writing Java code in a custom data action subclass.

When the request contains some named event, and the action mapping has a forward defined with the same name as the event, the data action will return this forward provided that:

- no errors occured during execution of the page lifecycle, and
- no forward has been set yet on the DataActionContext during execution of the page lifecycle.

JHeadstart has slightly extended this event-based forwarding. As explained in the section 'JHeadstart onEvent methods and the eventValue parameter', JHeadstart might use the same event name for multiple events fired by the same page. The eventValue request parameter will then hold the name of the actual action binding that must be executed. To allow for different page navigation based on the actual action binding that is executed, JHeadstart first looks for a forward with the same name as the value of the eventValue request parameter.

So, together with the "polymorphic" default forwarding as explained in the section 'ADF Data Action and Data Forward Action', the JhsDataAction and its superclasses will sequentially apply the following rules to determine the forward that defines to which page or action to dispatch:

1. If errors occurred during action processing, and the input property is set on the action mapping, the value of the input property will be returned as forward.
2. Respect user setting: if during the execution of the page lifecycle a forward is set on the DataActionContext, this forward will be used.
3. If the request contains a parameter named eventValue and the action mapping has a forward defined with the same name as the value of the eventValue parameter, this forward will be returned.
4. If the request contains a parameter named event and the action mapping has a forward defined with the same name as the value of the event parameter, this forward will be returned.
5. If the parameter property is set on the action mapping and the value is not "unknown", the value of the parameter property will be returned as forward.
6. If there is a forward named "success" defined for the action mapping, this forward is returned.
7. If one of the action mappings has the 'unkown' property set to "true", Struts will forward to this action mapping.

8. If none of the above rules apply, you will get a blank page!

JHeadstart PageLifecycleFactory

As explained before, the implementation of the lifecycle phases is different for UIX and for JSP and the actual logic of each phase is coded into View-specific lifecycle classes.

The methods in the lifececycle classes perform a callback to the corresponding method in the `DataAction` class to facilitate customization. When extending or overwriting default ADF behavior, you can override the method in the `DataAction`, you do not have to worry about the lifecycle classes because normally you will only use one view technology in your application.

Since the JHeadstart runtime code must support both JSP and UIX, and includes view-specific extensions, the JHeadstart runtime includes view-specific lifecycle subclasses `JhsStrutsJspLifecycle and JhsStrutsUIXLifecycle`, in addition to the data action subclass `JhsDataAction`.

It is quite easy to configure ADF to use your own lifecycle subclass: ADF leverages the plug-in functionality provided by the Apache Struts framework. In the `struts-config` you specify a so-called LifecylceFactory class using the `<plug-in>` tag that returns the proper lifecycle subclass:

struts-config.xml

```
207   <plug-in className="oracle.adf.controller.struts.actions.PageLifecycleFactoryPlugin">
208     <set-property property="lifecycleFactory"
209                   value="oracle.jheadstart.controller.strutsadf.action.UIXLifecycleFactory"/>
210   </plug-in>
211 </struts-config>
```

In the rare occasion that you would need to override a method in the lifecycle class, rather than in the DataAction class, you can use the same technique as JHeadstart to register your lifecycle class.

Reference: See the Javadoc of `UIXLifecycleFactory`, `JSPLifecycleFactory`, `JhsStrutsUixLifecycle`, and `JhsStrutsJspLifecycle`.

Multi-part Request handling

Struts has built-in support for handling multi-part requests. However, this support relies on using form beans: if Struts executes an action mapping that has an associated form bean that must be populated with the request parameters, Struts decodes the multipart request in order to populate the form bean.

This approach has two disadvantages:

- If actions are chained and both action mappings have a form bean associated, Struts tries to decode the multipart request again when populating the second form bean. Since the multipart request is already decoded, the second form bean will be populated with null values.
- It does not work when using UIX as view, because form beans are not used with UIX. The dataForm form bean that is specified through the name property of the action mapping when performing drag and drop is ignored in the UIX case,

because method `StrutsUixLifecycle.processActionConfig` nullifies the name property again on the action mapping, before Struts starts evaluating whether the action mapping has a form bean associated that must be populated.

Multipart requests do work with UIX because the UIX servlet decodes the multipart request in the Update Model phase when the `DataAction` calls to the UIX servlet to perform the multi update.

However, the fact that with UIX the multipart request is not decoded until the Model Update phase, has an important ramification: onEvent methods will no longer fire (bug 4034293). This is because ADF builds the event list from the request parameters, before the Prepare Model phase. Since the multipart request is not yet decoded, the event parameter is not recognized by the ADF code that builds the event list.

The JHeadstart runtime addresses both issues. The chained actions with separate form beans are handled properly because JHeadstart provides a subclass of the Struts multipart request handler class, `JhsMultipartRequestHandler`. This class ensures that the multipart request is decoded only once, and that the second form bean is populated using the same multipart request handler as the first one.

In the struts-config, the `JhsMultipartRequestHandler` class is specified in the controller tag:

```
<controller locale="true"
   processorClass="oracle.jheadstart.controller.strutsadf.JhsRequestProcessor"
   multipartClass=
          "oracle.jheadstart.controller.strutsadf.JhsMultipartRequestHandler"
/>
```

The timing issue with decoding the request for UIX is solved by JHeadstart in the `JhsRequestProcessor` class. In this class, a subclass of the standard Struts `RequestProcessor` class, method `processMultipart` has been overridden, which (indirectly) decodes the multipart request by populating a dummy Struts form bean. Furthermore, in class `JhsUIXPageBroker`, which extends the `UIXPageBroker` class, method `decodeMultipartRequest()` has been overridden to prevent UIX from trying to decode the multipart request again. Final the `_doModelUpdate()` method in `JhsInitModelListener` uses the Struts multipart request handler, to correctly update the model.

Reference: See the Javadoc of `JhsMultipartRequestHandler`, `JhsRequestProcessor.processMultipart()`, `JhsUixPageBroker.decodeMultipartRequest()`, and `JhsInitModelListener._doModelUpdate()`.

Request Submission

No hyperlinks (POST instead of GET)

In a typical transactional application, bookmarking URLs in the browser can be problematic. More often than not, an action of the user adds to a 'context' on the middle tier that is used and needed by following actions. It is therefore often not a valid option to try to create a 'snapshot' of a particular situation by bookmarking the current URL. When the user starts a new application session by closing and re-opening his browser and clicks on the bookmark, the URL will be passed to the application again, but as the 'context' is no longer available, the bookmark will probably not work and might produce erratic behaviour.

Although you can't prevent end users from creating bookmarks, application developers may limit the impact of bookmarked URLs by not using hyperlinks on buttons and links in the application pages, but by submitting HTML forms (with `method="POST"`) instead. This way, parameters such as event names are not visible in the URL bar of the browser and are therefore not bookmarked.

Every page generated by JHeadstart contains two HTML forms:

1. An HTML form called '`router`'. For UIX, this form is defined in the `standardPageLayout.uit` template. For JSP, this form is defined in `routerForm.jsp`, which is included in every JSP page. This form is used mainly for navigation.

2. An HTML form containing all the generated data items. Even when you have generated two or more groups on the same page, there is still one form. We will call this form the data form. This form is used for any actions that require end user input (such as 'Save' operations).

All buttons and links on JHeadstart-generated pages submit one of these two forms.

Efficient submission (reduced network traffic)

As explained in the previous section, there are two different HTML forms present in each page generated by JHeadstart. The router form is small and contains only a few hidden input items, like '`event`' and '`eventValue`' (more on this in section 'Navigation Events and Data Events'). The router form is submitted for record- or page navigation purposes, for instance when the user clicks on a tab, or on a 'next record'/'previous record' button. By submitting the router form, the data items in the page (which are contained in the data form) are not submitted. Especially in multi-row updateable pages where a significant amount of data might be stored in these items, this would have otherwise produced a lot of unnecessary network traffic. The data form holding the data items is only submitted when the user presses the 'Save' button on a page.

The navigation links and buttons in a page never submit the router or data form directly, but by calling the JavaScript functions '`doEvent()`' or '`doNavigate()`' (which submit the router form) or function '`doDataEvent()`' (which submits the data form).

Outstanding changes warning

When a user has made changes to any of the data fields on a page, but then clicks on a navigation link or button (causing the router form to be submitted) without pressing 'Save' first, his changes will not be posted to the application server and he will therefore lose them. Furthermore, any outstanding changes on the middle-tier will be rolled back, so that the new page or record starts 'clean', without remnants of an earlier, unfinished transaction. JHeadstart ensures he gets a JavaScript message when such a situation occurs, warning the user for the loss of the uncommitted changes. The user can either cancel out and save his changes, or press OK and continue with the navigation.

This check is invoked by the `doEvent` and `doNavigate` functions in `form.js` (the two JavaScript functions that submit the router form). The check itself is implemented in the `hasChanges()` function. This function loops over all forms in the document and checks if the user has changed the value of one or more of the data items, or if there are pending middle tier changes (indicated by hidden field `hasChanges = true`).

Two exceptions exist to this checking behavior:

1. The check is not performed for pages where changing the data is not possible. This is the case in Find Pages and in the Select page of a select-form layout. JHeadstart generates script into these pages that sets the global variable '`ignoreChanges`' to `true`.

2. Items that are not bound to the model are not checked. This is for example the case for the `searchAttribute` and `searchText` of a Quick Search region. Users may change these fields and navigate out of the page without getting a warning. For this type of items, a call to the JavaScript function `addToIgnoreChangedFields` is generated. This function adds the item name to the array `ignoreChangedFields` that holds the names of the items for which changes are ignored.

Reference: See the function `hasChanges()` in JavaScript library `form.js`. You can find `form.js` by going to your ViewController project, and opening `[HTML Root Directory]/jheadstart/form.js`.

Navigation Events and Data Events

This section will discuss in detail how the navigation links and buttons in the generated pages produce events that can be picked up by ADF and JHeadstart on the middle tier.

Hidden Form Fields

As explained earlier, all buttons and links end up submitting one of two forms in the generated pages: the router form or the data form. To allow events to be posted on the request, both of these forms hold the following hidden fields:

```
<input type="hidden" name="event"/>
<input type="hidden" name="eventValue"/>
```

ADF/JHeadstart events can be submitted to the middle tier by filling in these two hidden fields, '`event`' and '`eventValue`', and submitting the form that contains them. See section 'ADF Page Lifecycle - Process Page Events' to understand how ADF and JHeadstart use these request parameters.

Depending on the form and the layout type (UIX or JSP), these hidden fields come from different sources. For UIX, the router form (including these hidden fields) is included in the UIT template '`standardPageLayout.uit`', which is the base template of all generated pages. The data form, which is generated into the individual pages themselves, gets these fields from an import of the UIT template '`hiddenFormFields.uit`'. For JSPs, the router form (including these hidden fields) is included in each generated page by including '`routerForm.jsp`', and the hidden fields are included in the data form by importing '`hiddenFormFields.jsp`'. You can locate these UIT templates or JSP includes in the directory under the HTML root of your application that you specified in the Application Structure File as the value of Service property 'UIX/JSP Virtual Common Directory' (typically `/common/`).

Javascript API

Instead of writing JavaScript code in the '`onClick`' methods of the various buttons and links that interact with these fields directly, an intuitive JavaScript API is included in library '`form.js`' that takes away the need to code directly against these fields. This API consist of the following three functions:

- `function doEvent(event, eventValue, checkForChanges, rowKeyStr, doRollback)`

  This function acts on the router form, and will be used to send an event to be handled by the current Data Page. It will put the values that were supplied in the '`event`' and '`eventValue`' arguments into the corresponding hidden fields of the router form. Then, by default, it will check whether the user has made changes to any of the fields in the data form and warn the user of the fact that he is about to lose those changes in case he did. However, this behaviour can be prevented by providing boolean value '`false`' for the '`checkForChanges`' argument. The '`rowKeyStr`' argument and the '`doRollback`' argument are "advanced use", and should normally be left as default (by not supplying them). In both cases, a hidden field with the same name as the argument exists in the router form whose value will be filled based on the supplied argument value. The '`rowKeyStr`' field is empty by default; the '`doRollback`' field contains the value '`true`' by default, thereby rolling back any middle tier changes that might exist (see section 'Outstanding changes warning').

  Examples of typical use:

  ```
  javascript:doEvent('Create');
  ```
  Sends the '`Create`' event to the current Data Page, after checking for outstanding changes.

  ```
  javascript:doEvent('sort', 'EmployeeName');
  ```
  Sends the '`sort`' event, with eventValue '`EmployeeName`' to the server, after checking for outstanding changes.

  ```
  javascript:doEvent('exitStageLeft', null, false);
  ```
  Sends the '`exitStageLeft`' event, without performing the check for outstanding changes.

- `function doNavigate(actionUrl, event, eventValue, resetBreadcrumbs)`

  This function acts on the router form, and is used to navigate to a different Data Page, optionally specifying a specific event to be handled by that Data Page. Checking for outstanding changes will always be performed prior to navigation, and any middle tier changes will be rolled back when navigation is performed.

Examples of typical use:

```
javascript:doNavigate('Employees.do');
```
Navigates to the Employees Data Page

```
javascript:doNavigate('Employees.do', 'Create');
```
Navigates to the Employees Data Page, and triggers the execution of the 'Create' event on this Data Page (see section 'Create Handling').

- `function doDataEvent(event, eventValue, checkForChanges)`

  This function submits the data form, filling the supplied values for the 'event' and 'eventValues' arguments into the corresponding hidden fields in the form. The event will be posted to the current Data Page. The check for outstanding changes is by default NOT performed (as the entire data form holding all changes the user might have made is submitted to the middle tier, and will in all likelyhood be processed there. However, by supplying the boolean value 'true' for the 'checkForChanges' argument, this check may be forced.

  Examples of typical use:

  ```
  javascript:doDataEvent('Save');
  ```
  Sends the 'Save' event to the current Data Page.

  ```
  javascript:doDataEvent('advancedSearch', 'advancedSearch', 'true');
  ```
  Sends the 'advancedSearch' event and eventValue to the current Data Page, after performing the check for outstanding changes.

Reference: See the functions `doEvent()`, `doNavigate()`, and `doDataEvent()` in JavaScript library `form.js`. You can find `form.js` by going to your ViewController project, and opening `[HTML Root Directory]/jheadstart/form.js`.

Browser Back/Refresh Button Protection

JHeadstart ensures that your application is back-button protected: the user cannot save a new department twice by using the browser's back button or the browser refresh button after the department has been created successfully for the first time.

This is implemented by the following constructs:

- Each page (or more precisely, each HTML `<form>`) that must be back-button protected includes a hidden form field called pageTimeStamp. The value of this parameter is the session variable `lastIssuedPageTimeStamp`. If you want to remove the back-button protection for a specific page, you just need to remove one line from the generated page.
  For UIX remove the following line:
  ```
  <formValue name="pageTimeStamp" value="${lastIssuedPageTimeStamp}"/>
  ```
  For JSP remove the following line:
  ```
  <jsp:include page="/common/timestamp.jsp" flush="true"/>
  ```

- `JhsRequestProcessor.isPageTimeStampOk()` checks the `pageTimeStamp` parameter submitted with the request against the last issued page timestamp (using the method `isPageTimeStampOk()`. If the submitted page timestamp is not equal to the last issued page timestamp, the Struts `ActionForward` '`invalidPageTimestamp`' is set.
- The struts-config contains a global forward:
```
<forward
  name="invalidPageTimestamp"
  path="/common/BackButtonErrorPage.uix"
/>
```
(or `BackButtonErrorPage.jsp` of course)
- To ensure that the Back Button error is not raised when the user used a List Of Values (LOV) before returning to the page and submitting it, the LOV page must not update the last issued page timestamp. This is achieved by having the LOV submit the request parameter `pageTimeStamp="ignore"`. `JhsDataAction.isPageTimeStampOk` knows not to change `lastIssuedPageTimeStamp` in that case.

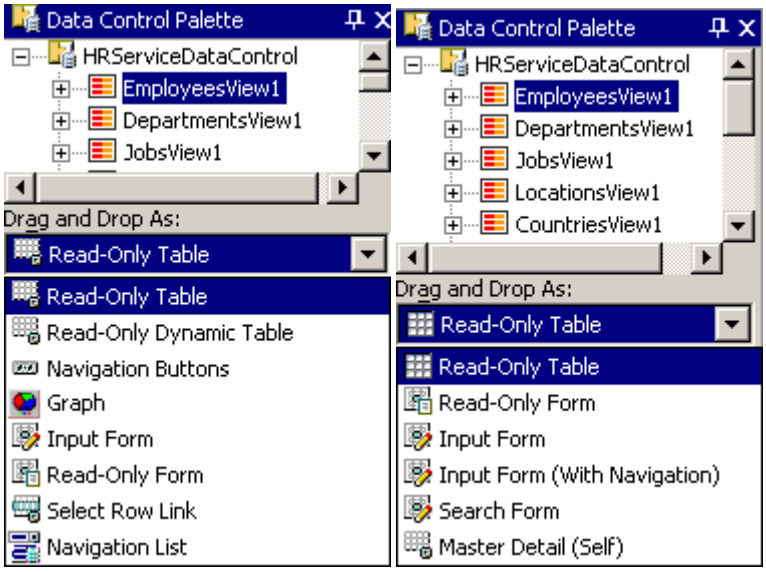
This provides back button protection. Pages that include the hidden `pageTimeStamp` field cannot be re-submitted after usage of the browser back button, or by using the browser refresh button.

Reference: See the Javadoc of the `JhsDataAction` methods `handleLifecycle()` and `isPageTimeStampOk()`.

Table

Multi-Row Insert/Update/Delete

If you want your ADF page to display multiple rows from a database table, the only drag-and-drop-as option is Read-Only. If you want to be able to update the data, you have to drag-and-drop as Input Form, which is a single-row layout.



Drag-and-drop-as choices for JSP and for UIX

JHeadstart has implemented generic multi-row update, insert and delete functionality for table layouts, both for JSP and UIX. This feature allows you to update and/or insert multiple rows, mark multiple rows for deletion and then save all these changes in one round-trip to the application server.

Section "Transactional Behaviors - Multi-Row Insert/Update/Delete" describes how this feature was implemented.

Sortable Tables

When creating a data bound table in ADF UIX, by default you can sort the rows by clicking on the header of an attribute. With JSP, a standard drag-and-drop data bound table is not sortable.

JHeadstart can generate JSP-tables that have sortable tables with similar functionality as in UIX. The runtime support for this is implemented in the `JhsDataAction.onSort()` event handler. This method is called when a form value named `event` is submitted that has the value `sort`. The `onSort()` method assumes that another form value named `eventValue` has as its value the name of the data binding of the ADF Business Components attribute. The ViewObject will be sorted by that attribute.

Reference: See the Javadoc of `JhsDataAction.onSort()`.

Detail-Disclosure

In some applications, a table row only displays a summary of properties, for a given element. In these cases it would be nice to have a way of clicking on a button and opening up a view with all the details pertinent to that row. The detail-disclosure feature of the UIX Table component provides this functionality.

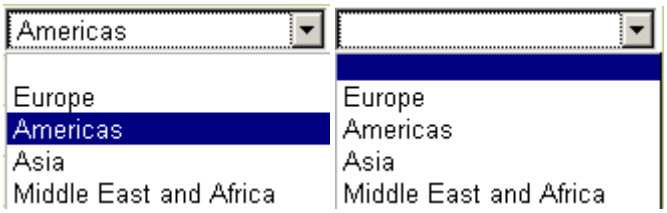
Generation: See Chapter 3 – 'JHeadstart Application Generator' for instructions how to generate detail-disclosure with JHeadstart.

The `JhsDataAction` class contains `onShow()` and `onHide()` methods that fire when the user clicks on the Show or the Hide button in a JHeadstart-generated UIX page This method gets the index of the row that the user chose and decides whether to disclose (or undisclose) depending on the event name. By default, when the user clicks on the show icon/link in a table row, the details of all other rows are hidden. If you want to keep the details of previously disclosed rows visible, you should include a request parameter named hideOthersOnShow with value "false" in the request (for example through a hidden field).

Reference: See the Javadoc of `JhsDataAction.onShow()`

Select List (Dropdown)

If you specify a Single Select List for an attribute (JSP: `<html:select>`; UIX: `<messageChoice>`) you may want the option to choose an empty or null value.



With JSP it is easy to do, just add another <option/> tag that does not have any value. With UIX it's more difficult to achieve. You could of course create a special Data Collection that includes an empty row, but that would mean misusing the Business Service (Model) layer, for example by writing a special database query just for this purpose.

JHeadstart generates Single Select Lists with a null entry, based on ADF BC View Objects, without changing the ADF Business Components themselves.

This feature is implemented in UIX using several runtime components. These components rely on the fact that if a UIX page needs a Select List, then the page will include a Table Binding that contains the options to be shown in the Select List (and not a List Binding, as you would get when you drag-and-drop an attribute as messageChoice). For the example dropdown list shown in the screenshots above, the Table Binding is called `RegionsViewLookup`.

Wrapping the ADF Table Binding

JHeadstart has created a `TableBindingFactory` class that is able to take a Table Binding, and add an empty row on top of it. In the `JhsDataAction.handleLifecycle()` method, this factory is stored on the request under the name `jhsTableBindings`, so that it can be accessed using EL (Expression Language).

Then, in the UIX page component for the select list (`<messageChoice>`), the child data of the select list is defined as follows (using the example of the RegionsViewLookup):

```
childData="${jhsTableBindings.RegionsViewLookup_1T.rangeSet}"
```

As you can see, the EL expression refers to `jhsTableBindings`, which is the JHeadstart `TableBindingFactory`. The part of the expression that says "`.RegionsViewLookup_1T`" tells the table binding factory to look for a Table Binding called "RegionsViewLookup", and "`_1T`" tells it to add one (1) row on Top (if you want the empty row at the bottom, specify "`_1B`"). The `rangeSet` property of a table binding exposes the rows in the current range of the view object as a collection (that is standard ADF functionality).

It sounds a bit like dirty programming to specify the number of empty rows and their position through a suffix to the Table Binding name. It would have been preferable if that information could have been passed properly using request attributes, but with UIX we cannot set a request attribute through EL the way we can with JSP using the `<c:set>` tag. That's why this solution was chosen: it works!

Reference: See the Javadoc of `JhsDataAction.handleLifecycle()`, `TableBindingFactory`, `TableBindingWrapper` and `RangeSetWrapper`.

Attention: The same `TableBindingFactory` is also used for other purposes: see "Transactional Behaviors - Multi-Row Insert/Update/Delete" and "Transactional Behaviors - Create Handling".

List Of Values (LOV) - UIX

UIX has a built-in component `<listOfValues>` that represents a LOV dialog window, complete with search and select facilities. This component is designed to work with the `<messageLovInput>` component, which is used as a text field for launching a List Of Values modal window. The LOV is launched when you click the flashlight icon (that is, if you specify the right destination Struts action in `<messageLovInput>`).

The `<messageLovInput>` component is similar to the `<messageLovField>` component, but the `<messageLovInput>` component supports partial page rendering (PPR), field level validation, and a multi-select LOV table. The latter two features are explained in more detail in the following sections.

Generation: See Chapter 3 – '*JHeadstart Application Generator*' for instructions how to generate Lists Of Values with JHeadstart.

Use LOV for Validation

The `<messageLovInput>` component fires a `lovValidate` event when the text input field loses focus and its value has changed. This is intended as a hook for checking the entered text to see if it is valid input, and if there is a unique match with a value in the LOV. By default this event is not handled, you will have to write the code to do that yourself.

If you checked 'Use LOV for Validation' when generating your LOV with JHeadstart, JHeadstart handles the `lovValidate` event for you in the `JhsDataAction.onLovValidate()` method. Just before firing this event, a JavaScript function specified in the `onLovValidate` property of the `<messageLovInput>` is called, in which JHeadstart passes as request parameters some values that are needed by `onLovValidate()`, like the typed in text for example.

If the typed in value is sufficient to uniquely identify a row of the LOV, the following things happen:

- The matching row is automatically determined.

- The underlying key of this row is copied to the corresponding value binding of the base page.
- ADF BC will automatically update the lookup attribute bindings (provided you based the lookup attributes on an Entity Association). You can verify this in the ADF BC Tester, if you update the base value attribute you should immediately see the changed lookup attributes.
- UIX's Partial Page Refresh feature shows the new value of the updated bindings in the page.

If there is no unique match, the LOV window is opened by setting request attribute `showLovWindow = true`. This is picked up by the `<messageLovInput>` element that has property `showWindow="${requestScope.showLovWindow}"`.

Attention: No JavaScript is used to copy the selected value to the page field; instead the selected value is used to update the data binding (by submitting a request to the application server).

Suggestion: You can add additional lookup fields to the View Object, which you can show in the base page. If the LOV is used to update the base value attribute, the lookup attributes will be automatically updated as well (verify that in the ADF BC Tester). You only have to make sure that Partial Page Rendering will refresh them, by adding the id of those fields to the partial target of the `<messageLovInput>` component.

LOV Window

When the LOV window is opened, UIX fires a '`lovFilter`' event. JHeadstart handles this event in `JhsDataAction.onLovFilter()`. If this happens just after the '`lovValidate`' event occurred, JHeadstart uses the typed in value of the `<messageLovInput>` field to perform a Search in the LOV. For example, if the user typed in 'foo', and the LOV contains multiple rows that start with 'foo' (case-insensitive), a search is performed so that only the rows starting with 'foo' are shown. (If there had been only one row starting with 'foo', then the '`lovValidate`' event would already have selected that value without opening the LOV.)

More about the way JHeadstart performs searches can be found in section 'Query Behaviors'.

When a user selects a row in the LOV, the JHeadstart event handler `JhsDataAction.onLovSelect()` is called. This method distinguishes between 3 kinds of places from which the LOV can have been called:

1. Search Area (in that case a "Find" binding needs to be updated)
2. Table Layout (distinguishes between an update and a new row that must be created, and distinguishes between single-select and multi-select LOV)
3. Form Layout (always an update of an existing row)

The type of LOV call is passed using the request parameter '`lovUsage`', specified in the destination attribute of the `<messageLovInput>` component. Depending on the type of LOV call, several other request parameters are picked up by `JhsDataAction.onLovSelect()`, to write back the selected value to the appropriate data bindings.

Reference: See the Javadoc of `JhsDataAction.onLovSelect()`.

Attention: No JavaScript is used to copy the selected value to the page field; instead the selected value is used to update the data binding (by submitting a request to the application server).

Suggestion: You can add additional lookup fields to the View Object, which you can show in the base page. If the LOV is used to update the base value attribute, the lookup attributes will be automatically updated as well (verify that in the ADF BC Tester). You only have to make sure that Partial Page Rendering will refresh them, by adding the id of those fields to the partial target of the `<messageLovInput>` component.

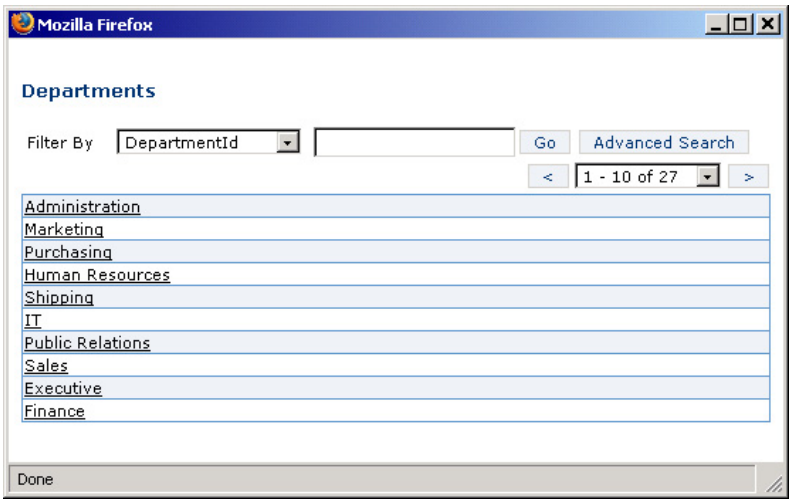
Multi-Select LOV

The `JhsDataAction.onLovSelect()` event handler knows that an LOV is multi-select when the request parameter `multiSelect=true`. This request parameter is set in the destination attribute of the `<messageLovInput>` component. In that case the `onLovSelect()` method will update the current row (on which the LOV was called) with the first selected value, and create new rows for the other selected values.

Reference: See the Javadoc of `JhsDataAction.onLovSelect()`.

List Of Values (LOV) – JSP

Unlike UIX, JSP does not have a built-in component for a List Of Values (LOV) dialog window. JHeadstart compensates that by generating a special JSP page for an LOV, and using JavaScript functions to call the LOV and to return the selected value. The generated JSP page can include Quick Search and Advanced Search. More about the way JHeadstart performs searches can be found in section 'Query Behaviors'.



Generation: See Chapter 3 – 'JHeadstart Application Generator' for instructions how to generate Lists Of Values with JHeadstart.

The LOV is called using the JHeadstart JavaScript function '`AdfJspLovPopup`'. This function opens the LOV JSP page in a separate dialog window.

When the user selects a row in the LOV, the JavaScript function `chooseLovValue` is called, which updates the current row in the page where the LOV was invoked.

In both cases the JavaScript calls can be found in JSP Includes (`lovPopup.jsp` and `lovSearchResult.jsp`), not directly in the generated pages.

Attention: Unlike the UIX implementation, with JSP JavaScript is used to copy the selected value to the page field; the data binding is not updated until you press the Save button.

Reference: See the functions `AdfJspLovPopup()` and `chooseLovValue()` in JavaScript library `form.js`. You can find `form.js` by going to your ViewController project, and opening `[HTML Root Directory]/jheadstart/form.js`.

Calendar

UIX has a built-in calendar popup that is automatically present when you use a `<messageDateField>` tag. Using this popup you can "browse through time" to select the date you want. For JSP, there is no such built-in component, so JHeadstart provides this functionality by shipping an HTML/JavaScript based calendar component provided by Sun. After running the "JHeadstart Enable Project Wizard", you will find the source files for this component in the "`jheadstart/calendar`" folder under your project's HTML root directory.

This calendar is integrated into the generated JSP pages by means of a jsp include after each date or datetime field. This is the '`calendarPopup.jsp`' file that is located, as all other jsp includes, in the '`commons`' folder under your project's HTML root directory. This include will render the little calendar icon that you see behind the HireDate field above. When clicked upon, the JavaScript function '`calendarPopup`' is invoked,

passing in the location of the calendar component, the item that the chosen date should be copied back into, and the date format mask that is taken from the 'dateformat.properties' resource bundle and therefore dependent on the locale of the user. It will invoke the calendar popup as shown above, which will allow you to navigate to and select the date you need.

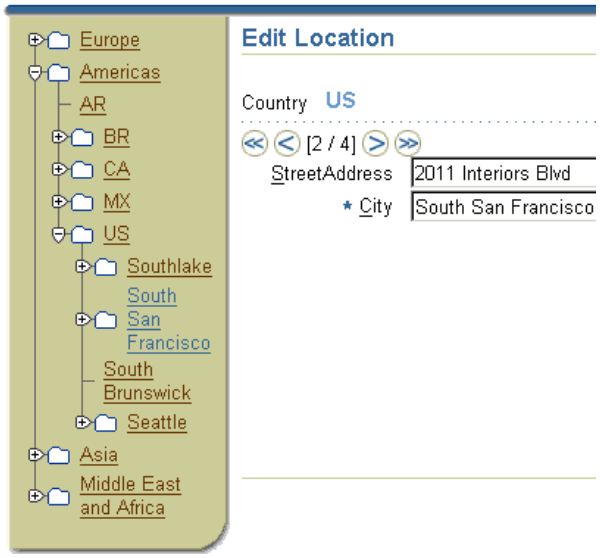
Known restriction:

The calendar component is able to return the selected date correctly for most date formats. It is, however, often not possible to for the calendar to recognize the date already present in the field. As this would often result in the calendar opening on totally arbitrary dates, we have made a small modification to the calendar component so that it always opens on the current date.

Reference: See the JSP file 'calendarPopup.jsp' in 'commons' directory under the project's HTML root, and the function calendarPopup() in JavaScript library form.js in the 'jheadstart' directory also under the HTML root. The sources for the calendar component itself can be found in the 'jheadstart/calendar' folder.

Tree

JHeadstart supports the generation of a data tree in UIX pages. The tree can be used to navigate to the row you want to see.



Generation: See Chapter 3 – 'JHeadstart Application Generator' for instructions how to generate trees with JHeadstart.

The JDeveloper Online Help, in particular the Oracle ADF UIX Developer's Guide, explains how to implement a tree in a UIX page, using the <tree> component.

To make a tree interactive (that is, the user can expand and collapse nodes of the tree), you need an object which implements the TreeDataProxy interface, which can be set as an attribute on the <tree> component. JHeadstart provides a useful implementation of the TreeDataProxy named JhsClientStateTreeProxy. It will manage the

expand/collapse state of nodes, as well as the selection on the tree both on the client and over the server, on a per render basis.

JHeadstart Runtime includes a Java class `TreeUtils` that allows you to have more than one tree in your application. By making sure that for each tree a separate *named* `JhsClientStateTreeProxy` object is available, you will never have the problem that if in the first tree the 10th node was expanded, it will also try to expand the 10th node of the second tree (and if the second tree does not have a 10th node, you would get an `ArrayIndexOutOfBoundsException`).

Reference: See the Javadoc of `TreeUtils`.

Refresh

If the number of tree nodes (or child nodes of a tree node) increases, the tree needs to be refreshed. This is required to prevent an `ArrayIndexOutOfBoundsException` in the `ClientStateTreeProxy` because the changed tree node would return more rows than expected by the existing proxy.

There are two types of situations where such a refresh is needed:

1. When a new row has been added to one of the tree nodes, or an existing tree node moves to another parent node.

2. When the tree shows only the children of a currently selected parent row, and the parent row changes. In this situation the parent is always maintained / selected on a separate page (not the tree page).

Ad 1.) JHeadstart handles the first type in `JhsDataAction` by calling the method `checkRefreshTree()` from the `onCommit()` event handler when the commit has been successful. It is not possible to detect in a generic way that one of the two abovementioned situations has occurred, therefore `checkRefreshTree()` always calls `onRefreshTree()`. In your own application you can override `checkRefreshTree()` and make the check more specific.

In `onRefreshTree()` the tree root iterator binding is reset, and the tree proxy object is removed from the session, causing the tree to be re-rendered in collapsed state. The name of the `TreeDataProxy` object to remove from the session is taken from request parameter "`treeProxy`". If you do not want the tree to be collapsed after an update because the row cannot be re-parented in the tree page, you can include a request parameter collapseTreeOnUpdate with value false.

You can add a button "Refresh Tree" on the tree page that causes `onRefreshTree()` to be executed, if you set the event property of the button to "`RefreshTree`".

Reference: See the Javadoc of `JhsDataAction.checkRefreshTree()` and `onRefreshTree()`.

Ad 2.) For the second type, the data action that handles the parent page needs to know which child tree(s) are dependent on this parent. As it is very difficult to recognize every situation where the current row of the parent might have changed, the child trees are refreshed each time the parent data action is called.

If you use JHeadstart's `JhsDataActionMapping` class for the struts-config action mapping, you can specify an extra property called '`treesToRefresh`' to the data action of the parent page:

```
<action path="…" input="…"
    type="oracle.jheadstart.controller.strutsadf.action.JhsDataAction"
    className="oracle.jheadstart.controller.strutsadf.action.JhsDataActionMapping"
    parameter="…">
  <set-property property="modelReference" value="…"/>
  <set-property property="treesToRefresh" value="[treeList]"/>
</action>
```

where *`[treeList]`* is a comma-separated list of dependent tree proxies, for example `"EmployeesTree, Employees2Tree"`.

The `JhsDataActionMapping` property '`treesToRefresh`' is picked up in the method `JhsDataAction.removeTreeProxies()`. This method is called from `JhsDataAction.handleLifecycle()`.

Reference: See the Javadoc of `JhsDataActionMapping` and the `JhsDataAction` methods `handleLifecycle()` and `removeTreeProxies()`.

Node Selection

With JHeadstart you can select a tree node to bring up the Edit page for the corresponding row. The Edit page usually queries from a different iterator (view object usage) than the iterator (view object usage) used to display the tree, to enable having recursive trees without specifying a separate view object usage for every possible level of the recursion.

JHeadstart implements the tree node selection by generating the following `<link>` destination in the tree's `<nodeStamp>` element:

```
<link … destination="[StrutsAction].do?event=selectTreeNode&[iterPath/keyPath]"/>
```

The *[StrutsAction]* is the DataPage to which we want to navigate, and *[iterPath/keyPath]* can contain multiple pairs of request parameters for backwards compatibility, but in the production version of JHeadstart it will contain only one pair:

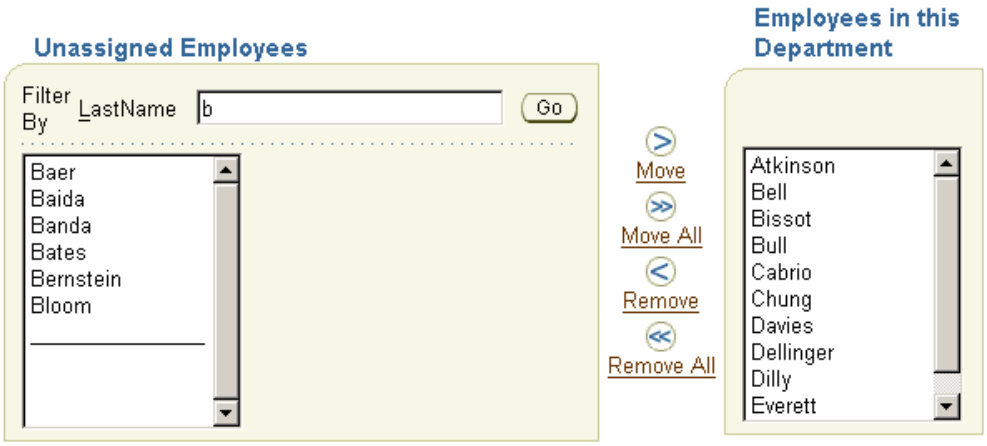
- `iterPath` refers to the Iterator data binding of the view object usage for the Edit page (for example `EmployeesIterator`, which refers to view object usage `EmployeesViewTreeSelect`)
- `keyPath` is the key value of the selected node (for example the selected `EmployeeId`)

Because the request parameter '`event`' has the value '`selectTreeNode`', the `JhsDataAction` event handler `onSelectTreeNode()` will be called, which interprets the `iterPath/keyPath` parameters and sets the current row in the specified iterator.

Reference: See the Javadoc of `JhsDataAction.onSelectTreeNode()`.

Shuttle

JHeadstart supports the generation of a shuttle in UIX pages. The shuttle provides a mechanism for moving items between two lists. Often the shuttle will be used to select items from one list by placing them in the other.



JHeadstart distinguishes two types of shuttles:

- Parent Shuttle: assigns existing rows to a (new) parent. In relational terms: updates the foreign key of existing records.
- Intersection Shuttle: creates new rows that are a link between two existing rows. In relational terms: creates new records in a table that has foreign keys to two other tables.

Generation: See Chapter 3 – '*JHeadstart Application Generator*' for instructions how to generate shuttles with JHeadstart.

When changes are made using the shuttle, and the page is submitted, UIX submits request parameters like `[shuttleName]:leading:items` and `[shuttleName]:trailing:items`, which contain a list of selected and unselected keys of the rows in the "leading" list (the left hand side) and the rows in the "trailing" list (the right hand side).

The left hand side can include a Quick Search area. See section 'Query Behaviors' for more information about the runtime handling of the Quick Search.

Shuttle Support in ADF BC Application Module

JHeadstart Runtime provides an extension for your Application Module that includes shuttle support. The `JhsApplicationModule` interface and the `JhsApplicationModuleImpl` class contain the methods `processParentShuttle()` and `processIntersectionShuttle()` that are able to analyze the list of selected and unselected items and translate that to updates and inserts to the database.

These methods are exported in the Client Interface of the Application Module, which makes them available as Data Control operations. For such an operation an Action Binding can then be created in the UI model of the shuttle page (which is of course what the JHeadstart Application Generator does).

The names of relevant View Object usages and attributes are defined in the UI model as parameters of the Action Binding. Using EL expressions, this is also done for the leading and trailing lists submitted by the shuttle (see above).

Reference: See the Javadoc of `JhsApplicationModule` and `JhsApplicationModuleImpl`, in particular the methods `processParentShuttle()` and `processIntersectionShuttle()`.

Processing Shuttle Operations

Data Control operations are normally called when a button is pressed that submits an event with the same name as the Action Binding in the UI model. In the case of a shuttle, however, the user normally presses the Save button after using the shuttle, while the Save button is only linked to the commit operation of the Application Module (and not to processParentShuttle or processIntersectionShuttle). The reason for that is that you might want to save other (non-shuttle) changes in the page at the same time.

So to correctly update the model with the shuttle changes, we need to explicitly call the shuttle processing Action Bindings when a shuttle page is submitted. JHeadstart does that in the `JhsDataAction.processUpdateModel()` lifecycle phase, where the method `JhsDataAction.processShuttle()` is called. This `processShuttle()` method checks if an Action Binding called `processParentShuttle` or `processIntersectionShuttle` is present in the UI Model, and if so, executes the method in the Application Module (passing the parameters that are partly taken from the request).

Reference: See the Javadoc of `JhsDataAction`, in particular the methods `processUpdateModel()` and `processShuttle()`.

Refreshing the Shuttle Lists

After a commit has been performed, the leading and trailing lists of the shuttle must be refreshed, to reflect the changes made by the user.

JHeadstart handles this in `JhsDataAction` by calling the method `refreshShuttleIterators()` from the `onCommit()` event handler when the commit has been successful. This method retrieves the names of the leading and trailing Iterator Binding from the shuttle processing Action Binding, and performs an `executeQuery()` on those iterators.

Reference: See the Javadoc of `JhsDataAction`, in particular the methods `onCommit()` and `refreshShuttleIterators()`.

File Upload / Download

File Upload

For attributes of types `BlobDomain`, `OrdImageDomain` and `OrdDocDomain`, JHeadstart can generate file upload items. In a generated page, you will get an input item of type '`file`'. When submitting such a page, you will send a multi-part request. See section 'HTTP Request Handling - Multi-part Request handling' for more information.

Depending on the attribute type you are uploading to, extra processing needs to be done. JHeadstart registers a customized ADF input handler to do the extra processing. The `JhsFormFileInputHandler` class is registered as handler for all the attributes of type

BlobDomain, OrdImageDomain or OrdDocDomain. This takes place as first step of the model update.

Reference: See the Javadoc of JhsDataAction.processUpdateModel(), JhsFormFileInputHandler.

File Download

For attributes of types BlobDomain, OrdImageDomain and OrdDocDomain, JHeadstart can generate file download items.

For OrdImageDomain and OrdDocDomain attribues, a call to the Intermedia servlet is generated, similar to drag-and-drop ADF.

For BlobDomain attributes, you will get a link with destination '/DownloadFile.do?pageTimeStamp=ignore&model=<uiModelName>&binding=<bindingName>&rowKeyStr=<currentRowKeyString>'. The DownloadFile class handles this action.

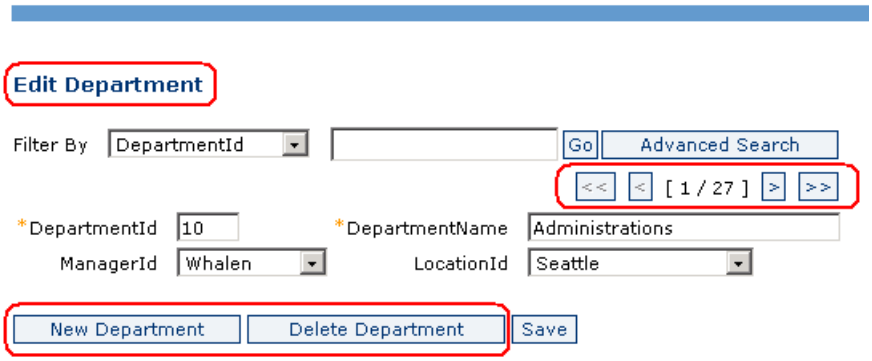
Reference: See the Javadoc of DownloadFile.

Same page for Insert/Update

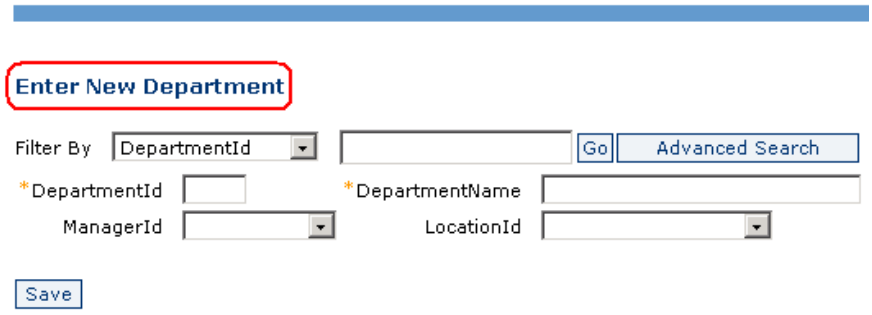
JHeadstart does not generate separate pages for handling the creation of new records, or the updating of existing records. Instead, one page is used for both situations. Depending on the page being in 'Create' mode or 'Update' mode, some elements on the page act differently:

1. The title of the page is 'Enter…' when in 'Create' mode and 'Edit…' when in 'Update' mode.
2. Items for record browsing are only shown in 'Update' mode.
3. New and Delete Buttons are only shown in 'Update' mode.

Page in 'Update' mode:



Page in 'Create' mode:



When pressing a 'New…' button, a '`Create`' event gets fired. This event is handled by the `onCreate` method of JhsDataAction. This method sets the request attributes '`createMode`' and '`createModeString`' to `true`.

In the page, elements that are dependent of the mode of the page, are rendered depending on the value of the `createMode` request attribute with conditional expressions appropriate for the View technology used:

- UIX: `<head title="${ui:cond(createMode,nls.INSERT_TITLE_DEPARTMENTS, nls.EDIT_TITLE_DEPARTMENTS)}">`
- JSP:
```
<c:if test="${createMode}">
  <jsp:include page="/common/heading1.jsp" flush="true">
```

```
      <jsp:param name="key" value="INSERT_TITLE_DEPARTMENTS"/>
    </jsp:include>
  </c:if>
  etc…
```

It is a bit different when you have two groups on the same page. In that case, the first group can be in 'Update' mode and the second in 'Create' mode. The request attributes set in the create handling for the second group are called '`createMode<groupName>`'.

Conditional expressions in UIX and JSP are generated accordingly.

UI Model Mapping

When building an ADF application with JDeveloper drag-and-drop, the UI Model (=binding container for a page) gets generated for you. By default the name of the generated UI Model is '`<pageName>UIModel`' where the pageName includes the relative path from the HTML Root.

As a consequence, when you have two pages referring to the same Data Collection (View Object), you have two binding containers, one for each page.

JHeadstart has only one binding container for each page that is generated for a group. For example when you generate a group with layout style = 'select-form' and an advanced search region on a separate page, you get a select page, a find page and a maintenance page. Each of these pages will reuse the same binding container. The name of the binding container is '`<groupname>UIModel`'.

For JSP, simply changing the `modelReference` parameter of the Struts action mapping is enough. However, the UIX `InitModelListener` cannot find the UI Model if the name is not '`<pageName>UIModel`'. Therefore, JHeadstart customized the `getBindingContainerName()` method in the `JhsInitModelListener`. It now retrieves the Struts `modelReference` parameter, and only if that does not exist it reverts to the page name algorithm.

Reference: See the Javadoc of `JhsInitModelListener.getBindingContainerName()`.

Query Behaviors

Query Bind Parameters

Sometimes you want the query of your view object to be dynamic: you want to be able to pass an argument (a bind parameter) to the where clause of the query. JHeadstart includes runtime support for passing such query bind parameters, defined in EL (the Expression Language).

Generation: See Chapter 3 – 'JHeadstart Application Generator' for instructions how to generate query bind parameters with JHeadstart.

Action Mapping property 'bindParams'

If you use JHeadstart's `JhsDataActionMapping` class for the struts-config action mapping, you can specify an extra property called '`bindParams`' to the data action.

You can pass query bind variables to any Iterator in your UI model, by adding the bold text to the struts-config action mapping of the relevant page:

```
<action path="…" input="…"
    type="oracle.jheadstart.controller.strutsadf.action.JhsDataAction"
    className="oracle.jheadstart.controller.strutsadf.action.JhsDataActionMapping"
    parameter="…">
  <set-property property="modelReference" value="…"/>
  <set-property property="bindParams" value="[iter1]=[expr1];[iter2]=[expr2]"/>
</action>
```

where you can specify one or more pairs of *`[iterX]=[exprX]`*, separated by '`;`'. *`[iterX]`* is the name of the iterator binding to which the bind parameters must be passed, and *`[exprX]`* is a comma-delimited list of literal values or EL-expressions that produce the values that must be passed as query bind parameters. For example: "`EmployeesIterator=King,10; DepartmentsIterator=Accounting`".

Reference: See the Javadoc of `JhsDataActionMapping`.

Processing of 'bindParams' in Data Action

The `JhsDataActionMapping` property '`bindParams`' is picked up in the method `JhsDataAction.applyIterBindParams()`. This method is called from two places: from the ADF lifecycle phase `prepareModel()` and from the end of `handleLifecycle()`, both in `JhsDataAction`.

The call from `prepareModel()` is needed because the first time an iterator rowset is used, ADF automatically performs a query in the `prepareModel()` method. If your query had bind variables that are not bound yet, an exception will occur. So the bind parameters have to be passed before the first query in `prepareModel()`.

The call from the end of `handleLifecycle()` is needed because during the lifecycle an update or an event might take place that changes the value of the bind parameter. For example a user has pressed the Next button for the parent of a shuttle, which should show a different list of available shuttle entries for the new parent. Before displaying the page again, the new bind parameters must be applied.

The method `applyIterBindParams()` extracts the `[iterX]=[exprX]` pairs from the `bindParams` property, calls an evaluator for the EL expressions, and checks the ViewObject of the relevant Iterator to see if the same where clause parameters have already been set. Otherwise it sets them, and executes the query of the View Object. The query is only executed if one or more of the bind parameter values were different from the old ones.

Reference: See the Javadoc of the `JhsDataAction` methods `prepareModel()`, `handleLifecycle()` and `applyIterBindParams()`.

Quick Search and Advanced Search

The JHeadstart functionalities Quick Search and Advanced Search share some runtime components: the Search Bean and the search support in the ADF BC Application Module.

Search Bean

One of the first things that are done when the `JhsDataAction.handleLifecycle()` method is called, is storing a special Struts form bean on the request under the name `"searchBean"`. This is done by the `JhsDataAction.storeSearchBean()` method. This Search Bean is used to hold the search values that a user typed in, and to remember if the user last did a Quick Search or an Advanced Search. The Search Bean of every page is stored in a Search Bean Map on the Session, so that when a user returns to the same page, the Search Bean of that page can be used to redisplay the previous search values.

The JHeadstart class `FindActionForm` implements the Search Bean. For UIX a subclass `UIXFindActionForm` is used, which extends `Map` so that we can use Expression Language (EL) to refer to the Search Bean in a UIX page (this is not necessary for JSP because we have tag libraries to directly refer to a form bean). Both classes extend the ADF `BindingContainerActionForm`, which exposes the binding values of the UI Model that is set on it at runtime. The Search Bean of a certain page is initialized with the UI Model of that page. `FindActionForm` however, only returns values that were previously populated from the request; it does not return the values of the underlying value bindings of the binding container, because we only want to remember the search values the user typed in. Furthermore, the `FindActionForm` class overrides the Struts superclass method `getDynaClass()` to return an instance of `JhsFindDynaClass`.

`JhsFindDynaClass` loops through all the value bindings of the UI model linked to the Search Bean, and prefixes their names with `"Find"` (to distinguish them from the normal value bindings of that name). For example, if a UI model contains a binding called `"LastName"`, the Search Bean will contain a `"FindLastName"`. This allows us to dynamically create the right form bean properties by specifying a UI model, without having to explicitly define a new form bean for each search page. The `JhsFindDynaClass` also adds a few fixed form bean properties:

- `searchAreaMode`
- `searchAttribute`
- `searchText`
- `showQuickSearch`

These are used to for the Quick Search (see separate section below), and for switching between Quick Search and Advanced Search.

Reference: See the Javadoc of the `handleLifecycle()` and `storeSearchBean()` methods of `JhsDataAction`, and the classes `FindActionForm`, `UIXFindActionForm`, and `JhsFindDynaClass`.

Search Support in ADF BC Application Module

JHeadstart Runtime provides an extension for your Application Module that includes advanced search support (which is used for both the Advanced Search and the Quick Search functionality of JHeadstart). The `JhsApplicationModule` interface and the `JhsApplicationModuleImpl` class contain the method `advancedSearch()` that takes an array of JHeadstart `QueryAttribute` objects and translates them to an additional where clause on the relevant View Object.

This method is exported in the client interface of the Application Module, which makes it available as a data control operation. For such an operation an action binding can then be created in the UI model of the page (which is of course what the JHeadstart Application Generator does when a Search Region is generated).

The name of the relevant View Object usage is specified in the UI model as parameter of the action binding (see also section 'Passing Action Binding Parameters'). Just before the operation is actually called, the array of `QueryAttribute` objects is constructed from the request and passed to the operation as well.

The `QueryAttribute` object stores information about that part of the search that applies to a single view attribute:

- attribute to search on
- operator to use
- search value
- format
- wildcard usage
- case (in)sensitivity

The `QueryAttribute` also translates the query operator names used in the pages by appropriate SQL operators and wildcard usage. For example, query operator "`startsWith`" results in the operator "`like`" and wildcard usage "`suffix`".

The `advancedSearch()` method uses this information to construct ADF BC ViewCriteria objects, applies them to the View Object, and then executes the (modified) query of the View Object.

Reference: See the Javadoc of the `advancedSearch()` method of `JhsApplicationModule` and `JhsApplicationModuleImpl`, and the Javadoc of the `QueryAttribute` class.

Calling the Application Module Search Support

Both Quick Search and Advanced Search call the `advancedSearch()` method of the Application Module by way of

`JhsDataAction.executeAdvancedSearchBinding()` (see previous section 'Search Support in ADF BC Application Module'). Two special situations are handled here:

1. The query returns no rows. A message is added and the Struts action forward is set to the input property of the action mapping (which is by default the page where the search was performed).

2. The number of rows exceeds request parameter `maxQueryHits`. An error message is added and the iterator binding is put in find mode, so no rows are shown. For more information about find mode, see section 'Auto Query'.

Reference: See the Javadoc of `JhsDataAction.executeAdvancedSearchBinding()`.

Quick Search

When you generate a Quick Search region using the JHeadstart Application Generator, you get the following components in your page:

- `searchAttribute`, which is either a hidden form field with as its value the single search attribute name, or a dropdown list of the attributes on which the quick search can be performed.
- `searchText`, field in which the user can type the search value
- "Go" button that submits a data event called quickSearch
- action binding in UI model for Application Module `advancedSearch()` method (used for both Quick Search and Advanced Search)
- hidden form fields (from the JSP include `hiddenFormFields.jsp` or the UIX template `hiddenFormFields.uit`) that specify the default way the search must be performed:

  1. `caseSensitiveQuery`, default value "`false`"
  2. `queryOperatorString`, default value "`startsWith`"
  3. `queryOperatorNonString`, default value "`is`"

Suggestion: You can override these settings at attribute level by setting the JHeadstart Custom ADF BC properties 'Query Operator' and 'Case Insensitive Query?'.

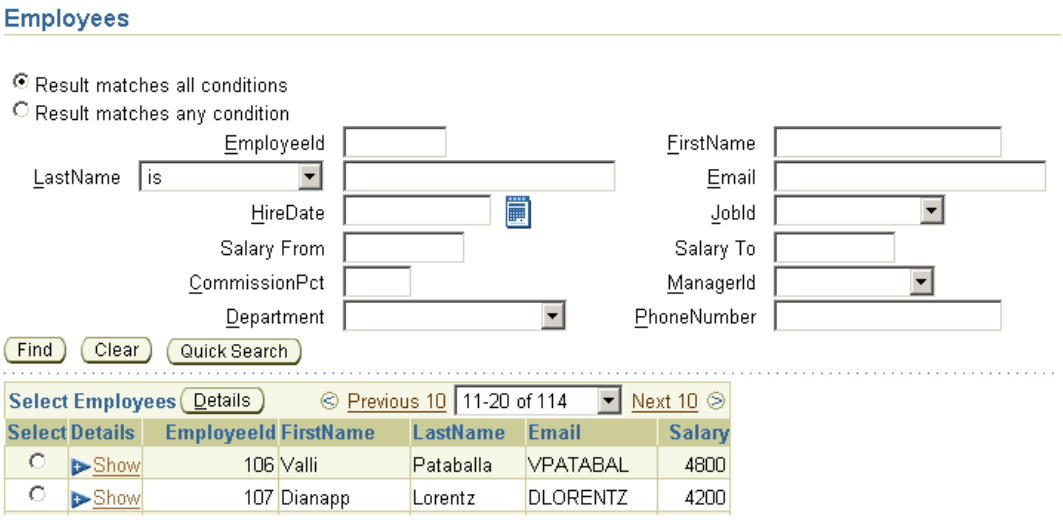
Executing the Quick Search

When you click the Go button of a Quick Search region, the event `quickSearch` causes the `JhsDataAction.onQuickSearch()` event handler to be executed.

The `onQuickSearch()` method takes the Search Bean (see section 'Search Bean') from the request and populates it with the request parameters submitted by the user. It then retrieves the `searchAttribute`, `searchText`, and the three relevant values from the `hiddenFormFields` (see above) from the Search Bean. These values are used to construct a single JHeadstart `QueryAttribute` object to pass to `JhsDataAction.executeAdvancedSearchBinding()` (see section 'Calling the Application Module Search Support'). The `QueryAttribute` object possibly overrides the Query Operator and the Case Insensitivity by looking at the Custom JHeadstart ADF BC properties.

Reference: See the Javadoc of `JhsDataAction.onQuickSearch()`.

Advanced Search



When you generate an Advanced Search region (choosing option 'samePage' or 'separatePage') using the JHeadstart Application Generator, you get the following components in your page:

- A radio group to select whether an “AND” or “OR” should be used in the SQL Where clause constructed from the query criteria.
- Fields for each queryable attribute, in which the user can type the search values, named `Find<groupName><attributeName>`
- "Find" button that submits a data event called `advancedSearch`
- “Clear” button that clears previously entered search criteria.
- action binding in UI model for Application Module `advancedSearch()` method (used for both Quick Search and Advanced Search)
- hidden form fields (from the JSP include `hiddenFormFields.jsp` or the UIX template `hiddenFormFields.uit`) that specify the default way the search must be performed:

    4. `caseSensitiveQuery`, default value `"false"`

5. `queryOperatorString`, default value "`startsWith`"

6. `queryOperatorNonString`, default value "`is`"

Suggestion: You can override these settings at attribute level by setting the JHeadstart Custom ADF BC properties 'Query Operator' and 'Case Insensitive Query?'.

Executing the Advanced Search

When you click the Find button of an Advanced Search region or Find Page, the event `advancedSearch` causes the `JhsDataAction.onAdvancedSearch()` event handler to be executed.

The `onAdvancedSearch()` method takes the Search Bean (see section 'Search Bean') from the request and populates it with the request parameters submitted by the user. It then passes the Search Bean to the `createArgumentListForAdvancedSearch()` method.

`JhsDataAction.createArgumentListForAdvancedSearch()` returns an `ArrayList` of `QueryAttribute` objects from the Search Bean with entered search criteria. Processing is as follows:

- Loop over all the control bindings in the current binding container
- Skip control bindings that are not a queriable value binding
- For each queriable value binding, check for the existence of a corresponding search bean value. When a search bean value exists, create a new `QueryAttribute`. Determine operator and case sensitivity based on custom properties and request parameters.

This list of search criteria is passed to `JhsDataAction.executeAdvancedSearchBinding()` (see section 'Calling the Application Module Search Support').

Reference: See the Javadoc of the `JhsDataAction` methods `onAdvancedSearch()` and `createArgumentListForAdvancedSearch()`.

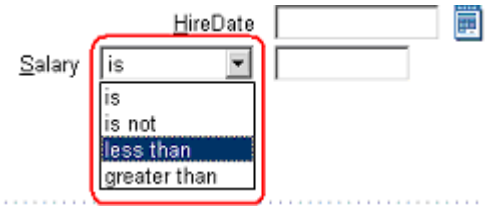
Query Operator

As described in the sections above, you can choose the Query Operator that is used. The default is specified in a hiddenFields include, and for specific attributes you can override it by setting the Custom JHeadstart ADF BC property 'Query Operator'. At runtime `JhsDataAction.createArgumentListForAdvancedSearch()` checks the ADF BC attribute definition to get the value of this custom property, and uses it to define a `QueryAttribute` object. See section 'Search Support in ADF BC Application Module' for more information about how the query operator is translated into SQL.

A special case is the Query Operator '`between`'. Then two input fields are generated for the same attribute:

In this case the request parameters suffixed with 'From' and 'To', for example 'FindEmployeesHireDateFrom' and 'FindEmployeesHireDateTo'. If JhsDataAction.createArgumentListForAdvancedSearch() sees that the corresponding ADF BC attribute has Query Operator 'between', it searches for these two request parameters and combines them into a single QueryAttribute object.

You can even let the application end user choose the Query Operator, by setting the operator to 'setByUser'.



The generated operator dropdown list will be called Find<groupName><attributeName>Operator, for example FindEmployeesSalaryOperator. If JhsDataAction.createArgumentListForAdvancedSearch() sees that the corresponding ADF BC attribute has Query Operator 'setByUser', it derives the operator from the chosen value in this additional dropdown list, which is submitted as a request parameter.

Reference: See the Javadoc of JhsDataAction.createArgumentListForAdvancedSearch().

Combining Quick Search and Advanced Search

If you generate both Quick Search and Advanced Search on the same page, by default the Quick Search region will be visible and the Advanced Search region will be hidden. Besides the normal Quick Search fields, there will also be a button called 'Advanced Search', to switch from Quick Search to Advanced Search.

When the user clicks the Advanced Search button, the Quick Search region is hidden and the Advanced Search region is shown, together with a button called 'Quick Search'.

Which search region is shown when initially showing the page, is governed by the Search Bean property 'showQuickSearch' (see section 'Search Bean'). The value is initially 'true', causing the Quick Search region to be shown.

When the button 'Advanced Search' is clicked, a JavaScript call simply switches the HTML class property of the relevant region to 'visible' or 'invisible', so there is no roundtrip to the application server.

When an Advanced Search is actually performed using the 'Find' button, the 'showQuickSearch' property of the Search Bean is changed to 'false'. The effect is that

when the user goes to a different page, and later returns to this same page, the Advanced Search region will still be visible. And vice versa, when a Quick Search is actually performed, the property is changed back to 'true'.

Auto Query

By default, JHeadstart generated pages automatically queries all rows, unless you generated with Auto Query turned off. Then records are queried upon request from the user, either by doing a Quick Search or an Advanced Search.

Generation: See Chapter 3 – 'JHeadstart Application Generator' for instructions how to generate auto query with JHeadstart.

Find Mode

JHeadstart's auto query functionality (or rather, the postponed query functionality) uses a feature of the ADF iterator binding: Find mode. When an iterator binding is set to work in Find mode, it switches to use a different row set iterator over a view criteria collection instead of the normal row set. This means that when Find mode is enabled, control bindings that reference the iterator binding will display attributes in the current view criteria row. Likewise, a table binding that references an iterator binding in Find mode allows you to render a table of current query-by-example view criteria rows. JHeadstart generated applications do not create any view criteria rows (because JHeadstart has its own advanced search features, see section 'Quick Search and Advanced Search'), therefore the effect of enabling Find mode is that no rows are shown.

When Find mode is disabled, the iterator binding switches back to work with its row set iterator over the data collection. This can be done explicitly by calling the iterator binding's `setFindMode()` method, or implicitly by calling its `executeQuery()` method.

The following sections explain in more detail how JHeadstart enables and disables find mode to implement postponed query.

Action Mapping property ' findModeIterators '

If you use JHeadstart's `JhsDataActionMapping` class for the struts-config action mapping, you can specify an extra property called '`findModeIterators`' to the data action. The value of this property should be a comma-delimited list of iterator bindings that exist in the UI model indicated by the modelReference property.

You can let any iterator of your UI model start in find mode, by adding the bold text to the struts-config action mapping of the relevant page:

```
<action path="…" input="…"
    type="oracle.jheadstart.controller.strutsadf.action.JhsDataAction"
    className="oracle.jheadstart.controller.strutsadf.action.JhsDataActionMapping"
    parameter="…">
  <set-property property="modelReference" value="…"/>
  <set-property property="findModeIterators" value="[iterList]"/>
</action>
```

where *`[iterList]`* is a list of one or more iterator binding names, separated by a comma. For example: "`EmployeesIterator, DepartmentsIterator`".

Reference: See the Javadoc of `JhsDataActionMapping`.

If you generate your JHeadstart application with Auto Query turned off for a certain group, then the iterator of that group's view object will be listed in the property 'findModeIterators'. If that group has a child group on the same page (and therefore is handled by the same Struts action mapping), then the child group iterator is also listed in that 'findModeIterators' property, so that when the page is first shown, neither parent nor child group show any rows.

Processing of 'findModeIterators' in Data Action

Normally, the first time a View Object's query is executed, is when a UI model containing an iterator binding for the View Object is prepared for the first time. This happens in the ADF Lifecycle method `prepareModel()`. So if JHeadstart wants to prevent that, the iterator binding's Find mode (see section 'Find Mode') must be enabled before the normal `prepareModel()` functionality is executed.

The `JhsDataAction.prepareModel()` lifecycle phase calls `JhsDataAction.setIteratorFindMode()` before the call to `super.prepareModel()`. The `setIteratorFindMode()` method looks at the `JhsDataActionMapping` property '`findModeIterators`' and only enables Find mode for the iterators listed there (if any).

The find mode is disabled again if either a Quick Search or an Advanced Search is performed, by calling `setIteratorFindMode()` again from `executeAdvancedSearchBinding()`.

Reference: See the Javadoc of the `JhsDataAction` methods `setIteratorFindMode()`, `prepareModel()`, and `executeAdvancedSearchBinding()`.

Transactional Behaviors

Multi-Row Insert/Update/Delete

The JHeadstart feature of multi-row insert, multi-row update and multi-select was already described in section 'User Interface Widgets- Table - Multi-Row Insert/Update/Delete'.

On OTN (Oracle Technology Network) you can find some JDeveloper How-To's that describe similar functionality: "How To Create Multi Row Edit Forms in a JSP Page " and " How-To Create an ADF UIX Editable Table". There are some drawbacks to these proposed solutions, however. The first one requires you to create new classes for each table you want to apply it to, and the latter only makes one row editable at a time. Also, none of these solutions provide multi-row insert and delete, only multi-row update.

This section describes how JHeadstart implements this feature at runtime without these drawbacks. For readability, we call the feature simply 'multi-row update' even though it also provides multi-row insert and delete.

Collecting the Changed Values for all Table Rows

Out of the box ADF is able to process the changed values of one row at a time. For multi-row updates we need to collect the changed values of multiple rows.

With UIX, you can easily do this using the UIX class `ServletRequestDataSet`, which extracts values from a ServletRequest's parameters (in particular from a UIX Component TableBean).

With JSP, it's more complicated. ADF uses a Struts Form Bean to pass the typed in values to the Model layer, and now we need an array of Form Beans. JHeadstart achieves this using the class `JhsBindingContainerActionForm`. Using this class you don't need specific Form Beans for each multi-editable table.

Reference: See the Javadoc of `JhsBindingContainerActionForm`.

Determining when to perform a Multi-Row Update

Multi-row update is performed when two conditions apply:

1. A table was submitted in the request, and
2. The method `JhsDataAction.doMultiRowUpdate()` returns true. This happens if one of the following conditions apply:
   - The request contains the `Commit` event (the Save button was clicked) , or
   - The request contains a parameter named '`multiRowUpdateEvent`' and the value of this parameter matches an event that is contained by this request. Note that you can include multiple events in the `multiRowUpdateEvent` request parameter, separated by commas.

Reference: See the Javadoc of `JhsDataAction.doMultiRowUpdate()`.

The actual checking of these two conditions is done in the JHeadstart UIX and JSP PageLifecycle classes.

Calling Multi-Row Update

In the JHeadstart specific PageLifecycle classes (see section 'HTTP Request Handling - ADF Page Lifecycle - JHeadstart PageLifecycleFactory'), one of the ADF lifecycle phases, `processUpdateModel()`, is overridden to add multi-row update processing. If the conditions for performing multi-row update apply (see previous section), a call is made to the `processMultiRowUpdateModel()` method in the same class. This method loops over all tables in the request, and calls `JhsDataAction.processMultiRowUpdateModel()` for each table. JSP tables are different from UIX tables (for example JSP uses Form Beans, and UIX doesn't), and that's why the code is in the PageLifecycle class instead of `JhsDataAction`.

In `JhsDataAction.processMultiRowUpdateModel()` all table request parameters (all values of the table rows) are analyzed. Read-only tables do not submit request parameters for the displayed values, so they pass quickly. For updateable tables, each row is inspected in turn. Rows that do not have a primary key yet, and have at least one attribute that is not default, will be inserted. The rows marked for deletion will be deleted. All other submitted rows are compared to the existing rows with the same primary key, and if at least one value is different, an update will be performed.

Reference: See the Javadoc of the `JhsStrutsUixLifecycle` / `JhsStrutsJspLifecycle` methods `processUpdateModel()` and `processMultiRowUpdateModel()`, as well as the method `JhsDataAction.processMultiRowUpdateModel()`.

Create Handling

Creating New Rows in a Form Layout

The '`Create`' event is fired when a 'New …' button is pressed. This event is handled by the `JhsDataAction.onCreate` method. Because multiple 'New…' buttons can exist in a page, the appropriate action binding must be found. The action binding is found based on the request parameters eventValue and event of the router form (see also section 'HTTP Request Handling - ADF Page Lifecycle - Process Page Events - JHeadstart onEvent methods and the eventValue parameter'):

- For the first 'New' Button (for creating a new parent) the `event` parameter has the value '`Create`'. The action binding with this name is executed.
- For the next 'New' button (for creating new children) the `event` parameter has the value '`Create`' and the `eventValue` parameter has as a value the name of the action binding to be executed.

After finding the action binding, the `binding.doIt() method` is invoked to execute the binding. In this case it means that a new row is inserted in the View Object associated with the binding.

After creating the new row, JHeadstart checks whether a default value needs to be calculated for some attribute in the View Object. A default value needs to be calculated for an attribute when the custom property DEFAULT_VALUE has a value for that attribute.

The `onCreate()` method also sets the `createMode` request parameter to true, which is necessary for reusing a page for insert and update, see section 'Same page for Insert/Update'.

Generation: See Chapter 3 – 'JHeadstart Application Generator' for instructions how to specify default value for an attribute.

As a default value can contain EL, the `Evaluator` class calculates the default value.

Reference: See the Javadoc of `JhsDataAction.onCreate` and the class `oracle.adf.controller.lifecycle.Evaluator`.

Creating New Rows in a Table Layout

When you want create functionality in a Table Layout, specify the **New Rows** property for that group. The generator will generate a number of empty rows at the bottom of the table where you can enter new data. The extra rows are not added to the View Object yet.

The functionality described in section 'Wrapping the ADF Table Binding' is used to add extra rows to the bottom of a table binding. The JHeadstart `TableBindingFactory` is stored on the request with for example the key `${jhsTableBindings.EmployeesView1_2B}`, meaning that two rows must be added at the bottom.

The `model` attribute of the generated (UIX) table refers to the `TableBindingFactory` as follows:

```
<table
  id="EmployeesView1"
  model="${jhsTableBindings.EmployeesView1_2B}".
```

For the rows added to the table, the default value of attributes is calculated when necessary.

When saving the data, JHeadstart performs a check whether the rows added to the table contain any changed attributes. When the new rows only contain default values, the row is not saved.

Reference: See method `processMultiRowUpdateModel` in `JhsDataAction`

Delete Handling

Delete in Form Layout

The '`Delete`' event is fired when a 'Delete …' button is pressed. This event is handled by the `JhsDataAction.onDelete()` method. Because multiple 'Delete…' buttons can exist in a page, the appropriate action binding must be found in the same way as described in section 'Create Handling'.

When execution of the delete binding is successful, the changes are automatically committed by calling onCommit(), see section 'Commit Handling'.

Reference: See the Javadoc of `JhsDataAction.onDelete()`.

Delete in Table Layout

In a table layout, a delete is triggered by checking the 'delete' checkbox in the table and pressing the Save button. The save button fires a 'Commit' event. JHeadstart detects that a multi-row delete takes place as described in section 'Multi-Row Insert/Update/Delete'.

Saving the State of Before the Delete

In both cases, before actually performing the delete, the current state of the application module is passivated for undo. Passivation can be compared to setting a savepoint in the database, however passivation takes place on the middle tier. When an error occurs later on, we can rollback the middle tier by activating the application state.

The reason is that if the delete fails because an error occurs during commit, then the row can be redisplayed to the user. If this was not done, the ADF Business Components would already have processed the delete and the row would not be visible anymore, even though the delete was still present in the database. See also section 'Commit Handling'.

Reference: See the Javadoc of the methods `passivateStateForUndo()` and `activateStateForUndo()` in `JhsDataAction`.

Commit Handling

Redisplaying Typed In Values when Error Occurs

Because of the multi-row update capability of JHeadstart, handling of validation failures had to be changed. When updating a table, it is possible that some changes are correct, while others violate business rules.

Imagine a table with two rows. The user marks the first row for deletion, and changes the second row. The deletion of the first row does not violate a business rule, but the update of the second row does. In this case, the first row is deleted from the middletier, while the second is still present. However, we want to present the page to the user with an error message indicating the business rule violation and **both** rows on the page, including the changes made to the second row. With standard ADF, this is not possible because the first row is deleted. An option would be to requery, but by doing so we would loose the changes the user made to the second row.

JHeadstart handles this as follows:

- The `JhsDataAction.processMultiRowUpdate()` method adds rows to be deleted to an `ArrayList`. They are not actually deleted yet. The inserts and updates are processed first.
- After that, it passivates the application module for undo (setting a kind of savepoint on the middletier).
- All the rows stored in the `ArrayList` of rows to be deleted, are actually deleted.

- The default ADF commit processing takes place.
- When a validation error occurs, the passivated application module state is activated (rollbacked).

Effectively, this means that deteles are processed after updates and inserts. When an error occurs, the state of the middletier is 'rolled back to savepoint', so the user will also see the deleted rows.

Reference: See the Javadoc of `JhsDataAction.processMultiRowUpdateModel()`, `JhsDataAction.handleLifecycle()`, `JhsDataAction.passivateStateForUndo()`, `JhsDataAction.activateStateForUndo()`, `JhsDataAction.processRowsToRemove()`.

Success/Failure Message

During commit processing three situations can occur:

- When the user tries to save data without having made any changes, he gets a 'No changes to save message'. JHeadstart checks the state of the application module. The message is added to the Struts action messages, which are put on the request. See also section 'Displaying Struts Action Errors/Messages in UIX'.
- When the commit succeeds, a 'Transaction completed' message is added to the Struts action messages.
- When the commit fails, the errors are placed on the request to display to the user.

Reference: See the Javadoc of `JhsDataAction.onCommit`, `JhsDataAction.StoreTransactionCompletedMessage()`, `JhsDataAction.StoreNoChangesToSaveMessage()`, `JhsStrutsUixLifecycle.reportErrors()`, `JhsStrutsJspLifecycle.reportErrors()`

Rollback Handling

When the user makes a change on a page that violates a business rule, he gets an error message. However, he has changed the model layer already. When the user corrects his error and successfully submits the change, everything is fine. However, the user can choose to leave the page without completing his change. In this case he leaves pending changes in the Application Module.

JHeadstart solves this. When the user navigates out of a page with pending changes, he will get a JavaScript warning. When the user still wants to leave the page, an `event_Rollback` parameter will be added to the request. This fires the `onRollback()` method in `JhsDataAction`. This method stores the keys of the current row of each View Object in the current binding container. After that, the rollback action binding is executed and the current rows are reset to the values before the rollback.

Reference: See the Javadoc of `JhsDataAction.onRollback()`, `JhsDataAction.storeRowCurrencies()`, `JhsDataAction.restoreRowCurrencies()`.

CDM RuleFrame Support

CDM RuleFrame is a Business Rule Framework that can be generated out of Oracle Designer to the Oracle Database. It is part of the Oracle iDevelopment Accelerators package, delivered by Oracle Consulting.

If you use CDM RuleFrame for your business logic layer implementation, you will need a special JHeadstart extension class `RuleFrameApplicationModuleImpl` for your ADF BC Application Modules.

Reference: See Chapter 3 'JHeadstart Application Generator', section 'Prepare Model for Generation' – 'Using CDM RuleFrame', to see how you can extend your Application Module to use RuleFrame.

The JHeadstart class `RuleFrameApplicationModuleImpl` extends in turn the default JHeadstart Application Module class `JhsApplicationModuleImpl`. The only difference is that another Database Transaction Factory will be used: `RuleFrameTransactionFactory`. This class makes sure that `RuleFrameTransactionImpl` will be used for all ADF Business Components transactions.

The class `RuleFrameTransactionImpl` contains the actual CDM RuleFrame customizations for opening a transaction, closing a transaction, and reporting error messages. It handles both the DML errors that can occur during the `postChanges()` method, and the RuleFrame errors that can occur during the `doCommit()` method.

Reference: See the Javadoc of `RuleFrameApplicationModuleImpl`, `RuleFrameTransactionFactory`, and `RuleFrameTransactionImpl`.

Internationalization

Resource Bundle Management

Struts has built-in support for using property files as message resource bundles. Message resource bundles can be used to make your application multi-lingual. If you do not have internationalization requirements, it is still useful to use message resource bundles to store “hard coded” text strings in a central place, where they can be easily found and maintained.

Reference: See Chapter 3 'JHeadstart Application Generator', section 'Internationalization', to see how you can ensure that your application supports the desired language(s).

Extended Struts Resource Bundle Management

The JHeadstart class `JhsMessageResources` extends the standard Struts `PropertyMessageResources` class. The JHeadstart message resources class provides you with the following additional functionality:

- Messages can be defined in Java classes that extend `java.util.ResourceBundle`, in addition to property files. Property files cannot handle multi-byte characters properly.
- You can have multiple "default" resource bundles instead of just one. This provides you with the flexibility to (re-)organize your resource bundles as you like, without the need to change `<bean:message>` tags in your JSP’s. Without this functionality, you would have to specify the bundle property in the `<bean:message>` tag when you want to use a non-default resource bundle.
- Already translated messages. If a message is stored as `ActionError` or `ErrorMessage` with an empty string as key, the first argument of the message, which should contain the translated message text, is returned as message text.

In addition, this class allows you to override JHeadstart and/or ADF Business Components messages. To do so, simply include the message key, for example JHS-00100 or JBO-27014 in one of the default resource bundles. If the key is not found in your default resource bundle(s), the standard JHeadstart or ADF BC message bundles are used.

To use the JHeadstart message resources functionality, you need the following in your struts-config:

```
<message-resources parameter="[comma-delimited list of resource bundles]"
   factory="oracle.jheadstart.view.strutsadf.JhsMessageResourcesFactory"
   null="false"/>
```

Reference: See the Javadoc of `JhsMessageResourcesFactory` and `JhsMessageResources`.

UIX Resource Bundle Handling

In UIX pages, there is no component similar to the `<bean:message>` tag of JSP. Therefore JHeadstart has created a class called `JhsBundleDataObject`, which extends `JhsMessageResources` (see previous section). This class implements the interface

oracle.cabo.ui.data.DataObject, which ensures that UIX pages can read data from the resource bundle.

In the generated UIX pages the resource bundle entries are referred to using ${nls.KEY_NAME}, where KEY_NAME is an existing key in one of the Resource Bundles. The term nls in these EL expressions refers to a JhsBundleDataObject that is stored on the ServletContext by the JhsActionServlet class. The JhsActionServlet should be specified in your web.xml file as the servlet class to run when a Struts action URL is requested.

Reference: See the Javadoc of JhsBundleDataObject and JhsActionServlet.initModuleMessageResources().

Date Format Handling

With standard ADF you have to make sure that for each date field, both the Business Service and the View layer apply the same date format. For example: if you have a date attribute in an ADF BC View Object, you have to set the date format property for that View attribute (or underlying Entity attribute) to ensure that the date is shown correctly in the page. With UIX, you can also specify an <onSubmitValidater> with a date format to ensure that the UIX calendar widget shows the dates in the same format, and that the UIX page gives a JavaScript alert if the date is entered in a different format.

If you don't specify exactly the same date format both in ADF Business Components and in ADF UIX, you won't be able to save any changes in a page that includes that date field. With JSP you only have to take care of the ADF Business Components date format, but you still have to do it for each date attribute separately. JHeadstart makes it easier for you to keep those date formats synchronized.

With JHeadstart you can specify the desired date format and datetime format for your whole application by changing just one file: dateformat.properties. You can even create locale-specific versions, like dateformat_en.properties and dateformat_fr.properties.

The JHeadstart Application Generator generates the contents of dateformat.properties (only the default file, no locale-specific versions) based on two settings in the Application Structure File: the Service level properties 'Date Format' and 'Datetime Format'.

The settings in dateformat.properties are picked up in the following places:

1. In the ADF Business Components they are used for rendering attributes of type Date using the right format. This is achieved by using a custom JHeadstart formatter class for each Entity attribute of type Date: JhsDateFormatter or JhsDateTimeFormatter. These classes also implement the technique described by Steve Muench in his article 'Working Around Bug 3958528 Where Date Format Truncates Time'.

2. In the generated UIX pages they are used in <onSubmitValidater> components of <messageDateField> for rendering the correct format in the UIX calendar and for validating user input when submitting the page. The UIX page refers to ${dateformat.datepattern} or ${dateformat.datetimepattern}. The term dateformat in these EL expressions refers to an object that is stored on the ServletContext by the JhsActionServlet class. The

JhsActionServlet should be specified in your web.xml file as the servlet class to run when a Struts action URL is requested.

3. In the JSP include calendarPopup.jsp, the datepattern or datetimepattern is taken from the Struts Resource Bundles (see section 'Resource Bundle Management') and used to return a selected date in the correct format.

Reference: See Steve Muench's article 'Working Around Bug 3958528 Where Date Format Truncates Time' at http://radio.weblogs.com/0118231/stories/2004/10/18/workingAroundBug3958528WhereDateFormatTruncatesTime.html.

Reference: See the Javadoc of JhsDateFormatter, JhsDateTimeFormatter, and JhsActionServlet.initModuleMessageResources(). Also see the JSP include calendarPopup.jsp, which can be found in your ViewController project under HTML Sources/common (assuming you use the System Navigator).

Character Encoding

JHeadstart dynamically changes the character encoding depending on the request locale (language setting). If you want to enforce a specific character set, for example UTF-8, you can add an init-param in the character encoding filter of your web.xml file as follows:

```
<filter>
  <filter-name>CharacterEncodingFilter</filter-name>
  <display-name>CharacterEncodingFilter</display-name>
  <filter-class>oracle.jheadstart.controller.CharacterEncodingFilter
  </filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
</filter>
```

In that case the request locale is ignored.

Reference: See the Javadoc of CharacterEncodingFilter.

Security

Introduction

Generally speaking, there are two popular ways to implement authentication ("who is the current user") and authorization ("is the current user allowed to do this") in J2EE web applications. The J2EE standards provide JAAS (Java Authorization and Authentication Services), which on the Oracle OC4J container is implemented by JAZN. It allows web developers to develop the security in their application totally independent of the chosen JAAS implementation, by using a simple API that can be invoked to answer security related questions such as "who is the currently logged in user" and "does this user belong to a specific 'role'". A great benefit from this approach is that the mechanism behind the retrieval of security information can be changed, for instance from file- or table based to LDAP based, without a single change in the application code itself. Furthermore, it is very convenient that during development, a simple file-based security mechanism can be, while in other environments such as test- and production environments, a fullblown security implementation such as LDAP can be implemented, again without any changes to the application code.

Another popular way of implementing security is through a 'homemade' security mechanism. Often this is implemented in application logic using ServletFilters. A common reason for implementing security this way is that the security information is stored in the same database as the application tables and is accessed using the 'Model' code of the application.

The guiding principle behind the security features of JHeadstart is that the way the application accesses the security information is as independent as possible from the chosen implementation (JAAS or custom security). This section will discuss what JHeadstart offers to facilitate implementing security in your applications.

Authentication Proxy

JHeadstart can generate security features into the application in two places: in the View layer by hiding tabs that the user is not authorized for, and in the Controller by using the standard Struts 'roles' property. This property prevents the execution of a Struts Action if the currently logged in user does not have access to any of the 'roles' listed in this property. By default, this Struts feature only works with JAAS based authorization.

Because we would like to implement security in the generated applications as transparently as possible with respect to the chosen implementation, JHeadstart has introduced a class called '`JhsAuthorizationProxy`'. An instance of this class is automatically created when the application is launched, and stored on the HTTP session. It will be invoked every each and every time the application needs authorization information. So whether JAAS is used or a custom security mechanism, and whether the information is needed in the View or in the Controller, this 'authentication proxy' is the single point that all authorization questions are being routed through.

The Authorization Proxy will determine whether standard JAAS, or a custom security implementation is used, and will forward the 'authorization question' accordingly. How it knows which security mechanism is used and how this mechanism is invoked will be

explained later on. For now it is important to onderstand that as far as the application logic in the View and/or Controller is concerned, there is no need to know how the security is implemented; it will simply communicate with the Authorization Proxy.

Reference: See the Javadoc of `JhsAuthorizationProxy`. The session-specific instance is created in the `storeUserRoles`() method of the `JhsActionServlet`; you caould override this method to use your own authorization proxy (sub)class.

Accessing the Authentication Proxy in the View layer

For implementing security features in the View layer, for instance hidding tabs and buttons or making fields read-only based on authorization information, it would be very convenient if the Authorization Proxy could be accessed through EL expressions. For that reason, the JhsAuthenticationProxy implements the Map interface, and is stored on the HTTP session with the key "jhsUserRoles". This means that if you, for instance, want a tab to be hidden if the current user does not belong to the 'admin' or 'manager' roles, you can use the following syntax (UIX example):

```
<link text="Admin Pages" rendered="${jhsUserRoles['admin,manager']}" …/>
```

Note that you can use a comma-separated list of role names. The Authentication Proxy will process them left-to-right until it finds a role that the current user belongs to, and return true in that case. If the user belongs to none of the roles, it will return false.

Accessing the Authentication Proxy in the Controller layer

As mentioned before, the Struts controller has built-in functionality to prevent the execution of an Action if the current user does not belong to one or more authorized roles. It has implemented this behaviour through the 'roles' property that you can set on an <action> element in the struts-config:

```
<action path="/adminpages.do"
    type="oracle.jheadstart.controller.strutsadf.action.JhsDataAction"
    className=".."
    parameter="…"
    roles="admin,manager">
```

When this property is set, Struts will invoke the standard J2EE method '`request.isUserInRole`(..)' on the HTTP Request to determine whether the currently logged in user belongs to any of the role names in the 'roles' property. If not, it will send a HTTP-400 error to the HTTP Response, for which you can optionally specify in the web.xml file of your application to which error page this error should redirect (otherwise the browser used by end user would display its own, generic error page upon receiving this error.

JHeadstart has overridden several methods the Struts `RequestProcessor` class the following functionality:

- Use the Authorization Proxy instead of calling the '`request.isUserInRole`(..)' method directly, to make it work both with JAAS and custom security implementations.

- Allow 'graceful' handling of authorization errors through an 'accessDenied' forward on the Action, so that the developer can specify for each individual action what should happen if an authorization failure occurs (rather than sending a generic HTTP Response error). For this reason, we needed to delay the check

for role access until the start of the 'processActionPerform()' method, because in the 'processRoles()' method which Struts uses for the authorization check, it is not yet possible to return an ActionForward. An example on how to use this feature:

```
<action path="/adminpages.do"
    type="oracle.jheadstart.controller.strutsadf.action.JhsDataAction"
    className=".."
    parameter="…"
    roles="admin,manager">
  …
  <forward name="accessDenied" path="/publicpages.do"/>
</action>
```

Reference: See the Javadoc of the methods processActionPerform(), processRoles() and checkRoles() in the JhsRequestProcesssor class.

Using JAAS based security

By default, the JHeadstart Authorization Proxy will redirect all security related questions to the standard J2EE API that will communicate with the JAAS implementation. This boils down to it invoking the methods getRemoteUser() and/or getUserPrincipal() on the HttpRequest, to obtain the current user, and invoking the method isUserInRole(..) to determine whether the current user belongs to a specific role. So when using JAAS, there is no custom code whatsoever needed in the application to get access to security information.

Note that if you attempt to use the security features of JHeadstart (i.e. use the Authorized Roles/Functions property in the Application Structure File, use the 'roles' property on your Struts Actions and/or use the 'jhsUserRoles' object in EL expressions) while you did NOT configure JAAS on your application server, all authorization questions will be answered by the JHeadstart Authorization Proxy with 'false', meaning 'not authorized'. This is done so that you can have a 'public part' in your application that does not require the user to be logged in at all, but will not show links to 'protected parts' of your application if he is not.

Using Custom security

As indicated before, the JHeadstart Authorization Proxy allows you to use either JAAS security or your own, custom security mechanism, without any changes in the application code. Also stated before is that by default, JAAS is used. So what do you need to do to have the Authorization Proxy use your own, non-JAAS mechanism instead? Well, not much. Basically, there are only two things you'll need:

1. A 'user context' object of some sort, that implements the JhsUser interface. This is a very simple interface to implement, as it has only two methods:

   public String getUserId()

   This method should return the (unique) username of the currently logged in

user.

`public boolean hasAccess(String functionName)`
This method takes a role/function name as its parameter and returns true or false, depending on whether the user belongs to the role (or has access to the function, whatever you want to call it). Note that while in the View and Controller layers comma-separated lists of roles/functions can be used, the `hasAccess()` method will only get invoked with a single role or function name. The Authorization Proxy will take care of parsing the comma-separated list and invoke the `hasAccess()` method on the JhsUser object once for each role/function, until either a 'true' is returned or it runs out of roles/functions.

2. The user context object implementing the `JhsUser` interface should be stored as an attribute on the HTTP session under the key 'jhsUser', immediately after (successfully) logging in.

The mere presence of an object on the HTTP session under the key "jhsUser", which implements the `JhsUser` interface, is sufficient for the Authorization Proxy to switch from using JAAS to using your 'user context' (`JhsUser`) object for answers to authorization questions. Every time the Authorization Proxy is approached to ask if the user belongs to a role or any of a number of roles, the `hasAccess()` method on the JhsUser object will be invoked, thereby making this object the 'single point of thruth' for all authorization questions.

Typically, the `JhsUser` object will be created in a ServletFilter, or in a Struts Action immediately after successfully logging in. JHeadstart offers a ServletFilter called `AuthenticationFilter`, which can be configured through <init-param> tags in the web.xml file to implement a typical login/check/success or fail/logout scenario using your own login page. See the JHeadstart Demo application for a reference implementation using security tables in the database that shows the usage of this filter.

One final, important note. A typical generated application using authorization features will often result in the '`hasAccess()`' method of the JhsUser object to be invoked many, many times during each HTTP request. This means that it should be implemented in the most efficient way possible. If expensive calculations or lookups need to be performed to calculate access for a specific role/function, consider caching the result so that in subsequent calls to the `hasAccess()` method, the result is readily available.

Reference: See the Javadoc of the `AuthenticationFilter class`.

Security in the Model

Until now, we've only been discussing security in the View and Controller. You might also need access to authorization information in the Model, for instance for business rule validation. As in the Model you do not by default have access to things like the HTTP Request, it is not possible to obtain the Authorization Proxy. So unlike in the View and Controller layer, it does matter whether you use JAAS, or your own custom security mechanism.

Using JAAS

In this case you can use the standard ADF BC methods to obtain JAAS information. These methods are implemented on the SessionImpl class. So from anywhere in ADF BC code where you have access to the (root) Application Module or the DBTransaction, you can have code like this to access the logged in user and it's authorized roles:

```
// First get the SessionImpl
SessionImpl session = (SessionImpl)this.getDBTransaction().getSession();
// Now we can get the JAAS username ..
String userName = session.getUserPrincipalName();
// .. and all the roles the user has access to..
String[] userRoles = session.getUserRoles();
// .. or just check access to a single role!
session.isUserInRole("admin");
```

Using custom security

This will mean that you have created a JhsUser object and stored it on the session. If you also use the JHeadstart AuthenticationFilter, it will make sure that this JhsUser object is set on every ADF BC application module your application uses by calling the 'setUser()' method. Now, you can access the JhsUser object (and thereby the answers to all authorization questions) anywhere in Business Components code by obtaining the (root) application module, and calling the '`getUser`()' method.

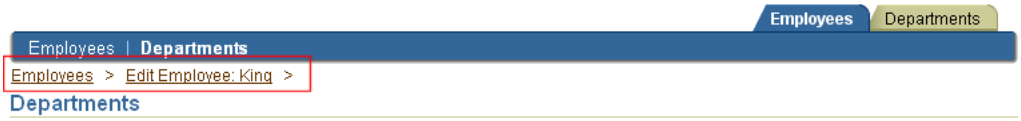
If you don't use the JHeadstart Authentication filter, you will need to implement this logic in a filter of your own. You can use the code in the' `setUserOnJhsAppModules`()' method of the `AuthenticationFilter`.

Breadcrumbs

Introduction

Breadcrumbs are the name for a 'list of previous steps' that is often displayed at the top of a page, that allow the user to 'safely' navigate back to earlier pages without using the browser's back button. Safely, because using the back button could bring you back to pages and requests that are in the middle of a transaction and may not be re-entered. Therefore, not every time a user clicks on a link or a button, a breadcrumb will be created.

Example of breadcrumbs in UIX:



In (almost) 100% generated applications, chances are you'll never need to know how the mechanism behind the breadcrumbs functionality works, and how you can influence its behaviour. However, if you are manually creating complex pageflows in a JHeadstart application, you might need to have some control over the way the breadcrumbs work. This section will explain this mechanism, and provide you with means to influence its behaviour to meet your requirements.

General mechanism

Every time Struts processed an Action, the list of breadcrumbs, called the breadcrumbs stack, is processed as well. The algorithm is roughly as follows:

1. There is one breadcrumb stack per struts-config file.

2. Only for Actions that use (a subclass of) `JhsDataActionMapping` as 'mapping' class can possibly make changes to contents of the breadcrumb stack.

3. If the forward that was returned by the Action navigates to a JSP or UIX page, a new breadcrumb is created and added to the stack.

4. A breadcrumb has two main properties: a destination and a label. When a new breadcrumb is created, the destination is set to the path of the Action that created it, so that clicking on the breadcrumb link will execute that Action. The label of a breadcrumb is the text that is shown to the end user in the page for this breadcrumb, and is therefore derived from the application's Resource Bundle (to allow for internatinalization). The algorithm to determine the label is quite involved and will be discussed later.

5. The same breadcrumb can only be present on the stack once, too prevent 'looping'. So if a breadcrumb is added to the stack which is already there, the stack is 'rolled back' to the breadcrumb BEFORE the one that is the same as the new one, and the new breadcrumb is added after it.

6. The 'identity' of a breadcrumb is not its label, but its destination: the URI that is fired when it will be clicked upon, which is the URI that will invoke the Action that created it.

7. If the forward returned by the current Action (see 2.) is NOT forwarding to a page but rather to another Action, the breadcrumb stack is searched to see if there is already a breadcrumb on it that was created by this Action. If so, its label is refreshed (more about labels later), which is necessary because it is possible to use dynamic labels that incorporate values from the bindings. If that is so, the execution of the Action might have changed the bindings, meaning the label should be 'refreshed' with the new binding values.

Reference: See the Javadoc and code of the classes `BreadcrumbManager`, `BreadcrumbStack` and `Breadcrumb`, and the `processBreadcrumbs()` method on the `JhsRequestProcessor` class for more details on the implementation of this algorithm.

Deriving the Breadcrumb label

The label of a breadcrumb is derived from the application Resource bundle. First, the 'base' key for the current breadcrumb is derived as follows:

```
BREADCRUMB_<Action path in uppercase without slash and without .do>
```

So an Action with path="/Employees.do" would have the 'base' key "BREADCRUMB_EMPLOYEES". This key is not immediately used to access the Resource bundle and obtain a translation for the label. First, we check the request for events. To understand why, consider the following. When the user clicks on a button labeled 'Create a New Employee', he will expect a breadcrumb labeled something like "Create Employee". Likewise, if he selects an existing employee and presses the 'Edit Employee' button, he'll expect a breadcrumb labeled "Edit Employee". However, in ADF, often the same DataAction is used for creating, updating, deleting and querying. As we learned before, the same Action means the same 'base' key, so it could not both be 'Create Employee' and 'Edit Employee'.

Therefore, as stated before, we first check the request for events. For every event on the request, we construct a new key based on the 'base' key and the event name. The Resource Bundle is checked for the existence of this 'base key + event' key. If it is not there, the next event is tried. If at the end no 'event-specific' entry is found, we simple fall back to using just the 'base' key.

To summarize, if we put the following lines in the application Resource Bundle:

```
BREADCRUMB_EMPLOYEES=Edit Employee
BREADCRUMB_EMPLOYEES.CREATE=Create Employee
```

then all breadcrumbs created by the '/Employees.do' Action will have a breadcrumb label 'Edit Employee', except when there was a parameter 'event=create' on the request when the breadcrumb was created, in which case the label will be 'Create Employee'.

Finally, it is possible to have 'dynamic' labels, by having the value in the resource bundle contain substitution parameters, like:

```
BREADCRUMB_DEPARTMENTS=Edit Department {0}
BREADCRUMB_EMPLOYEES=Edit Employee: ({0}, {1})
```

So how can we specify what the {0} and {1} parameters should be replaced with? We can do that by specifying EL expressions that will result in the substitution value, and pass them to the Action that created the breadcrumb through the 'breadCrumbParamExpressions' property in the struts config file:

```
<action path="/Departments.do"
        type=".."
        className="..">
  <set-property property="modelReference" value="DepartmentsUIModel"/>
  <set-property property="breadcrumbParamExpressions"
                value="${bindings.DepartmentName}"/>
```

The value of this property can be a comma-separated string of EL expressions, where the position of the Expression in the list corresponds with the number of the substitution parameter ({0}, {1} etc).

Influencing the Breadcrumb Stack behaviour

The standard behaviour of the breadcrumbs as described above is suitable 9 out of 10 times, especially with applications that are mostly 100% generated. When complex pageflows are created manually in JHeadstart applications, you might encounter situations where you need to change the default behaviour. To allow you to do this, we have added three breadcrumb-related properties to the `JhsDataActionMapping` that govern the behaviour of the breadcrumb stack.

The breadcrumbKey property

This property allows multiple DataActions to 'share' a single breadcrumb. As you will remember, the 'same' breadcrumb may only appear once on the breadcrumb stack. If a 'duplicate' breadcrumb is about to be added to the stack, the stack is first rolled back to just before the breadcrumb that is about to be duplicated, and then the new breadcrumb is added. Normally, two breadcrumbs are 'duplicates' if they have the same 'destination', ie.point to the same DataAction. Using the 'breadcrumbKey' property, you can specify that instead of the destination, the specified value for the 'breadcrumbKey' property is used to determine whether two breadcrumbs are duplicates. So in this scenario:

```
<action path="/Departments.do" …. >
  <set-property property="breadcrumbKey" value="mykey"/>
</action>
<action path="/Employees.do" …. >
  <set-property property="breadcrumbKey" value="mykey"/>
</action>
```

will result in the situation that either the 'Employees' breadcrumb or the 'Departments' breadcrumb can be present on the stack.

This might be very useful when, for instance, you have a tree structure like in the JHeadstart Tutorial. When you click around in the tree, selecting Regions, Countries, Locations and Departments in the tree, you don't want a list of breadcrumbs to be created. Basically, the entire tree should always result in just one breadcrumb: the one for the currently selected node. This can be achieved by putting the same 'breadcrumbKey' on all the DataActions for this tree, and, in fact, that is how JHeadstart will generate it.

The addBreadcrumb property

Any Action that uses the JhsDataActionMapping will, by default, create a breadcrumb when it is executed. In some cases, you might need an Action that uses this ActionMapping class, while it should not create a visible breadcrumb. You can do this by specifying the addBreadcrumb property on this Action, and set it to false (default is true):

```
<action path="/Departments.do" …. >
  <set-property property="addBreadcrumb" value="false"/>
</action>
```

Note that, in spite of the property name, a breadcrumb IS added to the stack, but it is 'invisible'; it won't be rendered in the breadcrumb stack. This means that if this action is executed and an (invisible) breadcrumb created by this action is already on the stack, the stack WILL be rolled back. Without providing an example, there are cases where you might need this. If you don't, take a look at the next property.

The rollbackBreadcrumbStack property

With this property, you can prevent the breadcrumbStack to be rolled back if the breadcrumb that is added to the stack already exists on the stack. Use with care, as it could result in mile-long breadcrumb stacks.

When used together with the 'addBreadcrumb' property, you can prevent an Action from modifying the breadcrumb stack altogether (which would be the same as using an ActionMapping that does not extend the JhsDataActionMapping):

```
<action path="/Departments.do" …. >
  <set-property property="addBreadcrumb" value="false"/>
  <set-property property="rollbackBreadcrumbStack" value="false"/>
</action>
```

Default value for this property is 'true'.

Changing the Breadcrumbs Stack appearance

If you want to change the way the breadcrumbs are rendered in your application, or perhaps you want to not render them at all, you'll be pleased to know that you won't have to edit each individual page in your application. You can make the changes in a single location, even after you finished generating. This location depends on whether you used UIX or JSP as the view layer

Rendering Breadcrumbs in UIX

In UIX, the <breadcrumbs> tag that renders the breadcrumbs is incorporated into the standardPageLayout.uit file. This UIX template, which is used by all pages, can be found in the 'commons' directory under your application's HTML root. You can make changes there to alter the appearance of the stack in all pages.

Rendering Breadcrumbs in JSP

When using JSPs, the code that determines how the breadcrumb stack is rendered is located in the 'breadcrumbs.jsp' file, which can be found in the 'commons' directory under your application's HTML root. It gets included into the generated pages because the <jsp:include> is present in all the 'page level' JJT (JHeadstart JSP Template) files, that are present in your project.