ORACLE JHEADSTART 10g for ADF

(RELEASE 10.1.3.3)

DEVELOPER'S GUIDE

JULY, 2008

ORACLE

JHeadstart Developer's Guide

Authors: Steven Davelaar, Peter Ebell, Ton van Kooten, Sandra Muller, Jaco Verheul

Contributors: Pieter Biemond, Sigrid Gylseth, Bouke Nijhuis

CONTENTS

CHAPTER

1 Getting Started

Developing complex transactional applications on the Java Enterprise Edition (JEE) platform is not a straightforward task. Java en JEE are widely perceived as a complex development platform with relatively low developer productivity. However, if you choose the right development tools, you will experience that this perception is simply not true. This developer's guide is about such a tool set, consisting of Oracle JDeveloper, Oracle's Application Development Framework (ADF) and Oracle JHeadstart. This toolset provides you with an unprecedented productivity and ease of use in building feature-rich JEE web applications in a flexible, and highly maintainable way.

To understand what we mean with unprecedented productivity, **we strongly recommend that you first go through the JHeadstart Tutorial**. This tutorial is the best way to get started with JHeadstart, it does not require any prior Java or ADF knowledge, and provides an excellent overview of the development process and main features JHeadstart brings to the table.

JHeadstart Tutorial - Building Enterprise JSF Applications with Oracle JHeadstart.
http://www.oracle.com/technology/products/jdev/tips/muench/jhstutorial

After you have completed the tutorial, you probably can't wait to build your own applications. The content of this developer's guide, together with numerous pointers to external sources provides you with everything you need to know to build enterprise-class web applications.

The first section in this chapter provides a brief introduction into the components and technologies of the toolset, with references to external sources that provide more information about each of the components.

The last section contains a comprehensive roadmap to build web applications with this toolset.

1.1. Introduction into JDeveloper, ADF and JHeadstart

To get the most out of JHeadstart, it really helps to understand more about the underlying technologies. If you have used Oracle Designer in the past to generate Oracle Forms aplications you probably agree that good knowledge of Oracle Forms is rather helpful in generating more complex functionality. This also applies to JHeadstart, understanding how technologies like ADF Data Binding, ADF Faces and JSF work, is indispensable for generating complex applications that involve customizations to the default generator templates used by JHeadstart. This section provides the pointers to obtain this knowledge.

1.1.1. Oracle JDeveloper

Oracle JDeveloper is the Integrated Development Environment (IDE) that allows us to work productively. It 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 Java EE development by providing wizards, editors, visual design tools, and deployment tools to create high quality, standard Java EE components including applets, JavaBeans, Java Server Faces (JSF), servlets, and Enterprise JavaBeans (EJB). JDeveloper also provides a public Add-in API to extend and customize the development environment and seamlessly integrate it with external products.

Oracle JDeveloper on OTN. Overview, Online Demo's, Tutorials, White Papers, How-to's, Feature list, and more:
http://www.oracle.com/technology/products/jdev

1.1.2. Oracle Application Development Framework (ADF)

Oracle ADF is an end-to-end J2EE framework, fully integrated with JDeveloper that simplifies development by providing out of the box infrastructure services and a visual and declarative development experience. Since it supports multiple technologies you have the choice to use the components that best fit your situation.

Oracle ADF 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 Java Server Faces 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.

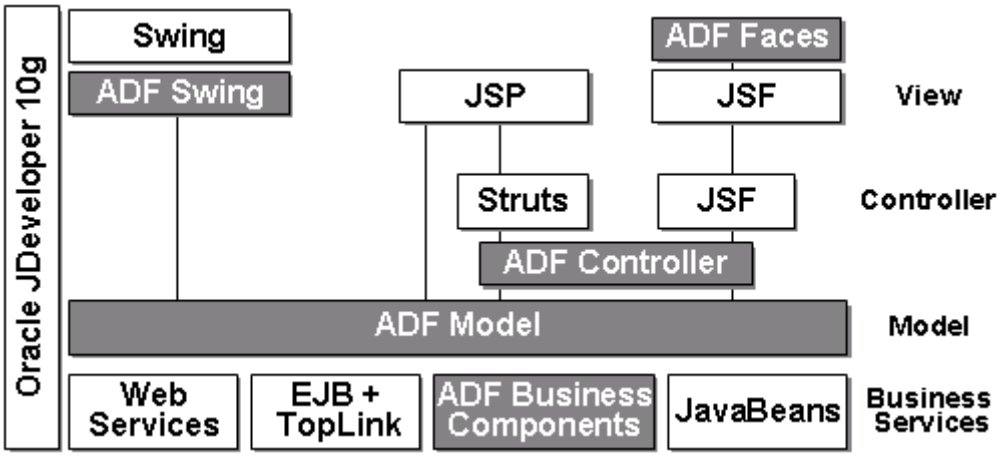


Figure 1-1 Oracle ADF Architecture

Oracle ADF on OTN. Overview, Online Demo's, Tutorials, White Papers, How-to's and more:
http://www.oracle.com/technology/products/adf

ADF Developers Guide for Forms/4GL Developers. Very comprehensive developer's guide with guidelines, best practices, hints and tips on building applications using ADF.

1.1.3. What is Oracle JHeadstart?

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

JHeadstart consists of three main components:

- JHeadstart Runtime Library

  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 (JSF config file) , View (ADF Faces pages), 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 Java EE 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 Forms2ADF Generator (JFG)

  In addition, JHeadstart offers you assistance in moving from the Oracle Forms world to the Java/J2EE world. Using the JHeadstart Forms2ADF Generator, the Forms .fmb files are read, and based on the Forms elements defined in the form, the JFG creates ADF Business Componenets, as well as the XML meta-data (Application Definition) required by the JHeadstart Application Generator. After running the JFG, you can then run the JAG to create a fully functional ADF web application, based on the definitions in the Oracle Form.

Oracle JHeadstart on OTN. Overview, Online Demo's, Tutorials, White Paper, How-to's and more:
http://www.oracle.com/technology/products/adf

JHeadstart Weblog. Tips and tricks, advanced techniques and how to's on ADF and JHeadstart.
http://blogs.oracle.com/jheadstart/

1.2. Roadmap to Developing ADF Applications using JHeadstart

This section provides you with a roadmap to build web applications using ADF and JHeadstart. Although the tasks and task steps are presented here as sequential, it is likely that you will iterate around some of these steps, refining the details at every pass of the iteration.

The roadmap provides a short description of each step, and references the section in this developers guide where you can find more information related to the task. As such, the roadmap can be seen as an extended table of contents of this developer's guide, although the scope is even broader. Some steps will refer to external sources, like the ADF Developers Guide or articles on the JDeveloper/ADF corners on Oracle's Technology Network (OTN).

Since this is a developer's guide, it does not include activities for project management, quality control, testing and user documentation. The nature, sequencing, and contents of these activities is pre-determined by the project approach (Waterfall, Iterative, Agile, DSDM, XP, etc.) and is beyond the scope of this guide.

1.2.1. Set Up Project for Team-Based Development

1.2.1.1. Setup Version Control System

Reference: Chapter 2 "Set up Project for Team-based Development", section "Setting up Version Control System"

1.2.1.2. Set up Structure of JDeveloper Application

Reference: Chapter 2 "Set up Project for Team-based Development", section "Set up Structure of JDeveloper Workspace and Projects"

1.2.1.3. Define Project Standards for Organizing ADF Business Components

Reference: Chapter 2 "Set up Project for Team-based Development", section "Defining Java Package Structure and Other Naming Conventions"

1.2.1.4. Define Java Package Structure and Other Naming Conventions

Reference: Chapter 2 "Set up Project for Team-based Development", section "Defining Java Package Structure and Other Naming Conventions"

1.2.1.5. Define Project Standards for Organizing JHeadstart Application Definition Files

Reference: Chapter 2 "Set up Project for Team-based Development", section "Organizing JHeadstart Application Definition Files"

1.2.2. Create Business Service using ADF Business Components

1.2.2.1. Create Business Component Base Classes

Reference: Chapter 3 "Creating ADF Business Components", section "Setting Up ADF BC Base Classes"

Web Reference: Oracle Application Developer Framework Developer's Guide for Forms/4GL Developers Release 10.1.3, chapter 25: *Advanced Business Components Techniques*, section 25.1. *Globally Extending ADF Business Components Functionality*, and section 25.2. *Creating a Layer of Framework Extensions*.: http://download-uk.oracle.com/docs/html/B25947_01/bcadvgen.htm#sm0291.

1.2.2.2. Create Entity Objects and Associations

Reference: Chapter 3 "Creating ADF Business Components", section "Creating the Entity Object Layer"

Web Reference: Oracle Application Developer Framework Developer's Guide for Forms/4GL Developers Release 10.1.3, chapter 6: Creating a Business Domain Layer Using Entity Objects. http://download-west.oracle.com/docs/html/B25947_01/bcentities.htm - sm0124

1.2.2.3. Create View Objects and View Links

Reference: Chapter 3 "Creating ADF Business Components", section "Creating View Objects and Application Modules"

Web Reference: Oracle Application Developer Framework Developer's Guide for Forms/4GL Developers Release 10.1.3, chapter 5: Querying Data using View Objects. http://download-west.oracle.com/docs/html/B25947_01/bcquerying.htm - sm0070

Web Reference: Oracle Application Developer Framework Developer's Guide for Forms/4GL Developers Release 10.1.3, chapter 7: Building an Updatable Data Model with Entity-based View Object. http://download-west.oracle.com/docs/html/B25947_01/bcvoeo.htm - sm0167

1.2.2.4. Create Application Modules

Reference: Chapter 3 "Creating ADF Business Components", section "Creating View Objects and Application Modules"

1.2.2.5. Implement Business Rules

Reference: Chapter 3 "Creating ADF Business Components", section "Implementing Business Rules"

Web Reference: Implementing Business Rules in ADF BC. White paper on OTN. http://www.oracle.com/technology/products/jdev/collateral/papers/10131/businessrulesinadfbctechnicalwp.pdf

1.2.3. Design and Generate Web Pages

1.2.3.1. Understand JHeadstart Generator Achitecture and Add Ins

Reference: Chapter 4 “Using JHeadstart”

1.2.3.2. Create Application Definition File

Reference: Chapter 4 “Using JHeadstart”, sections “Using the Create New Aplication Definiton Wizard”, “Using the Application Definition Editor”.

Reference: Chapter 2 “Set up Project for Team-based Development”, section “Organizing JHeadstart Application Definition Files”

1.2.3.3. Configure Internationalization Options

Reference: Chapter 11 “Internationalization and Messaging”, section “National Language Support in JHeadstart”

1.2.3.4. Generate and Run First-cut Web Application

Reference: Chapter 4 “Using JHeadstart”, sections “Running the JHeadstart Application Generator” an “Running the Generated Application”

1.2.3.5. Design and Generate Page Layouts

Reference: Chapter 5 “Generating Page Layouts”

1.2.3.6. Design and Generate Item Display Types and Item Behavior

Reference: Chapter 6 “Generating User Interface Widgets”

1.2.3.7. Configure Query Behavior in Pages

Reference: Chapter 7 “Generating Query Behaviors”

1.2.3.8. Configure Transactional Behavior in Pages

Reference: Chapter 8 “Generating Transactional Behaviors”

1.2.3.9. Design and Generate Menu Structure

Reference: Chapter 9 “Creating Menu Structures”

1.2.4. Design and Generate Security Structure

1.2.4.1. Understand and Choose Authentication and Authorization Options

Reference: Chapter 10 "Application Security", section "Understanding and Choosing Security Options with JHeadstart"

Web Reference: OC4J Security Guide http://download-uk.oracle.com/docs/cd/B25221_03/web.1013/b14429/toc.htm

Web Reference: Introduction into ADF Security in JDeveloper 10.1.3.2 http://www.oracle.com/technology/products/jdev/howtos/1013/adfsecurity/adfsecurity_10132.html

1.2.4.2. Implement User Authentication

Reference: Chapter 10 "Application Security", sections "Using JAAS-JAZN for Authentication"

Reference: Chapter 10 "Application Security", sections "Using JAAS with Custom Login Module for Authentication"

Reference: Chapter 10 "Application Security", sections "Using Custom Authentication"

1.2.4.3. Implement Role-based and Permission-based Authorization

Reference: Chapter 10 "Application Security", sections "Restricting Access to Groups based on Authorization Information"

Reference: Chapter 10 "Application Security", sections "Restricting Group And Item Operations based on Authorization Information"

Reference: Chapter 10 "Application Security", sections "Using Custom Authentication"

1.2.4.4. Design and Generate Security Administration Pages

Reference: Chapter 10 "Application Security", sections "JHeadstart Security Tables and Security Administration Screens"

1.2.5. Customize Generated Web Tier

1.2.5.1. Decide on Customization Approach

Reference: Chapter 10 "Using JHeadstart", section "Recommended Approach for Customizing JHeadstart Generator Output"

1.2.5.2. Use ADF Design-Time Tools to Implement Post-Generation Changes

Reference: Chapter 14 "JSF-ADF Page Lifecycle"

Web Reference: : Oracle Application Developer Framework Developer's Guide for Forms/4GL Developers Release 10.1.3, part III "Building your Web Interface". http://download-west.oracle.com/docs/html/B25947_01/partpage3.htm - sthref869

Web Reference: Oracle Application Developer Framework Developer's Guide for Forms/4GL Developers Release 10.1.3, chapter 10 "Overview of Application Module Data Binding". http://download-west.oracle.com/docs/html/B25947_01/bcdcpal.htm - sm0255

1.2.5.3. Move Post-Generation Changes to Custom Templates

Reference: Chapter 4 "Using JHeadstart", section "Customizing Using Generator Templates"

1.2.5.4. Create Custom ADF Faces Skin

Web Reference: Oracle Application Developer Framework Developer's Guide for Forms/4GL Developers Release 10.1.3, section 22.3 "Using Skins to Change the Look and Feel". http://download.oracle.com/docs/html/B25947_01/web_laf003.htm#CACJAGIG

Web Reference: ADF Faces Skinning Selectors. http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/doc/skin-selectors.html

Web Reference: ADF Faces Skin best practices. http://emarcoux.blogspot.com/2007/03/adf-faces-skin-best-practices.html

1.2.5.5. Add New Items and Customize Generated Items at Runtime

Reference: Chapter 12 "Runtime Page Customizations"

This page is intentionally left blank.

CHAPTER

2

Set Up Project for Team-Based Development

This chapter provides guidelines on

- selecting and setting up a version control system
- setting up the structure of the JDeveloper Application
- organizing JHeadstart application definition files
- defining the Java package structure and other naming conventions

2.1. Setting Up Version Control System

Good version control is indispensable when working in teams. There are many version control systems available on the market. In this section we will provide guidelines and recommendations for setting up version control. The following topics are discussed:

- Version control models
- Requirements for a good version control system
- Which files to version?

2.1.1. Version Control Models

When selecting a version control system, you have to choose between two basic models of version control: file locking and version merging. Wikipedia provides the following definitions for these two models:

- **File Locking**: The simplest method of preventing concurrent access problems is to lock files so that only one developer at a time has write access to the central "repository" copies of those files. Once one developer "checks out" a file, others can read that file, but no one else is allowed to change that file until that developer "checks in" the updated version (or cancels the checkout).

  File locking has merits and drawbacks. It can provide some protection against difficult merge conflicts when a user is making radical changes to many sections of a large file (or group of files). But if the files are left exclusively locked for too long, other developers can be tempted to simply bypass the revision control software and change the files locally anyway. That can lead to more serious problems.

- **Version Merging**: Most version control systems, such as CVS and SubVersion, allow multiple developers to be editing the same file at the same time. The first developer to "check in" changes to the central repository always succeeds. The system provides facilities to merge changes into the central repository, so the improvements from the first developer are preserved when the other programmers check in.

  The concept of a *reserved edit* can provide an optional means to explicitly lock a file for exclusive write access, even though a merging capability exists.

Revision Control in Wikipedia. Overview and definitions
http://en.wikipedia.org/wiki/Revision_control

For developing applications using JDeveloper, ADF and JHeadstart, we recommend to use the version merging approach, for the following reasons:

- It is a file-oriented development environment. Even for small to medium-sized applications, you will easily have hundreds of files to manage. It is inevitable that at some point multiple developers need to modify the same files. Using the File Locking approach this means that developers will have to wait for each other to finish a task and check in again. Although the number of “locking conflicts” can be reduced by a smart distribution of development tasks, it is our experience that you can never entirely avoid it.

- The files that most often are modified simultaneously by multiple developers are XML files, which by its structured nature are very well suited for automatic merging by version control systems. It is our experience that a version control system like SubVersion is also very good at merging Java files, the other most used type of file in this development environment.
- When generating your application using JHeadstart, many files are modified during a generation run. When using the file locking approach, you need to know upfront which files will be modified by the JHeadstart Application Generator: all these files need to be checked out prior to generation, otherwise they remain read only and will not be modified by JHeadstart. It requires in depth knowledge of JHeadstart and ADF to be able to correctly "predict" which files will be modified in a specific generation run. With the version merging approach this is not an issue, once you have finished a development task, the version control system will tell you which files have been modified and need to be committed to the version control repository.

2.1.2. Requirements for a GoodVersion Control System

When selecting a version control system, make sure the system provides functionality to address the following requirements

- It supports the *Version Merging* model (see previous section).
- It provides a so-called *Atomic Commit*. With this we mean that when you have modified a number of files that you want to commit to the version control repository, you want either the entire transaction to be committed, or the entire transaction to be rolled backed when a version conflict is detected which cannot be solved by an automatic merge. In other words, either all files are committed succesfully, or none of the files are committed. A version control system like CVS does not support an atomic commit. This means that some files might be committed to the repository, and then a version conflict is detected and the rest of the files cannot be committed. When this happens, you end up with an inconsistent situation in your version control repository since there are many interdependencies between files in an ADF environment. Obviously, when other developers update their local copies with this inconsistent set of files, they are likely to run into all sorts of problems and error messages.
- It detects file changes by comparing file content rather than the timestamp of the file. This requirement is particularly important when using JHeadstart: when you regenerate your application using the JHeadstart Application Generator, the content of many files might remain the same, although the file is recreated with a new timestamp. When you commit your work after you completed a development task, you do not want a new version of all these unmodified files to be committed to the repository. Otherwise, it will be really hard to find back versions that contained a real change, being a version you might want to revert to when you want to undo some work.
- An efficient and easy to use user interface to perform common versioning tasks. Developers should spend as little time as possible with version control tasks. An intuitive user interface for common tasks like updating their local copy, commiting changes, reverting to previous versions, resolving merge conflicts, and creating application releases is essential to meet this requirement. Ideally, the versioning user interface is integrated with JDeveloper, although in our experience it is not a big deal to switch with Alt-Tab to a stand-alone GUI for versioning when JDeveloper integration is not available, or less feature-rich.

A popular open source version control system that meets all of the above requirements is SubVersion (also known as SVN) . SubVersion has been built by the same community that is responsible for CVS. It is intended as a replacement for CVS, keeping all the good things of CVS, and fixing the bad things (like the absence of an atomic commit).

TortoiseSVN is an excellent stand-alone SubVersion GUI for the Windows platform, nicely integrated with MS Windows Explorer. JDeveloper integration is also available.

SubVersion Home Page. Overview, documentation and download.
http://subversion.tigris.org/

TortoiseSVN Home Page. Overview, documentation and download.
http://tortoisesvn.tigris.org/

Using JDeveloper with SubVersion. Developers guide and installation instructions.
http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/subversion/subversion.html

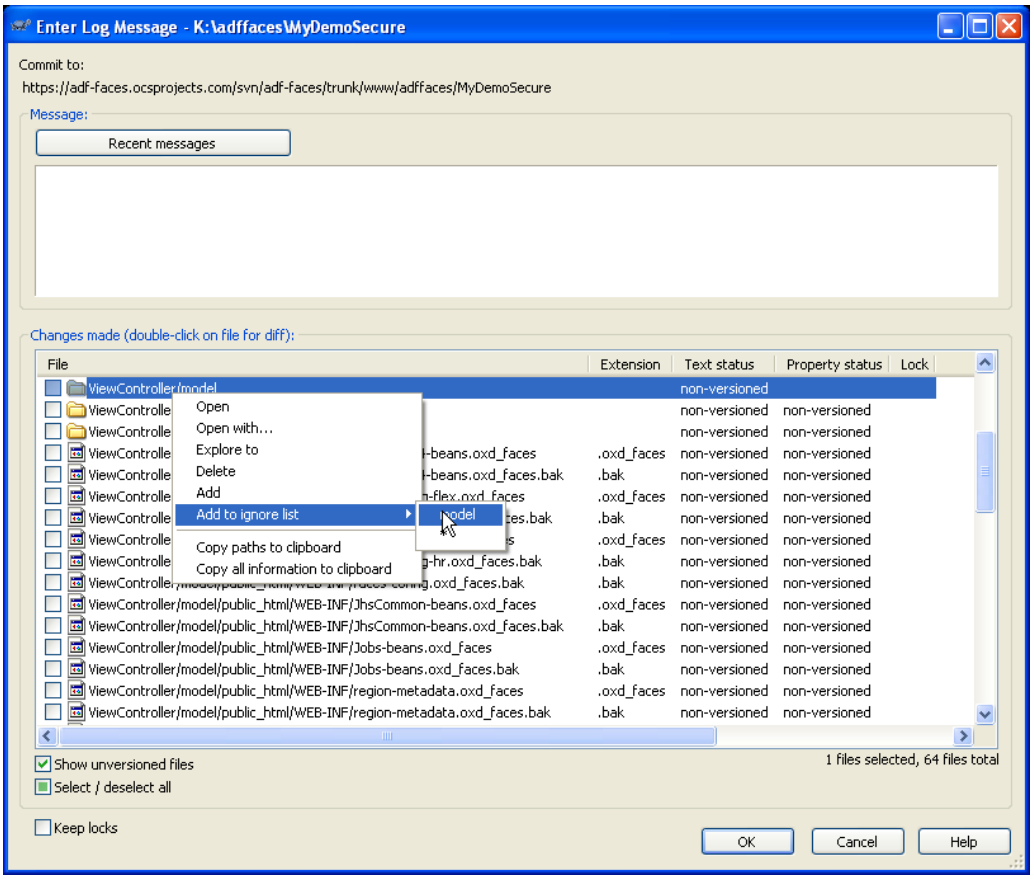
The JHeadstart team has succesfully used SubVersion and TortoiseSVN on a number of projects. This does not imply you should make the same choice. Version control is no rocket science, any system that meets the above requirements will do the job.

2.1.3. Which Files to Version?

We recommend to version all files in your project, except for

- derived files like all compiled Java classes and XML and property files that are copied to the classpath. In SubVersion, the easiest way to exclude these files is by adding the root directory of the classpath (typically the /classes directory) to the ignore list. This can be done by a right-mouse-click on the folder, and then choose Tortoise SVN -> Add to Ignore List …
- files in the temporary directory created by ADF Faces. When running your application in JDeveloper, a temp directory will be created under the WEB-INF directory, which holds cached ADF Faces files like images and stylesheets. This directory is not required to run your application and does not need to be versioned.
- the files created by JDeveloper in the root: appname-data-sources.xml, appname-jazn-data.xml, appname-oc4j-app.log, appname-oc4j-app.xml and application.log. If you are using JAAS-JAZN authorization, you do might want to version appname-jazn-data.xml since it holds the users and roles you have defined in JDeveloper for the project.
- the faces-config diagram files, with extension ".oxd_faces". When generating your application with JHeadstart, the faces-config diagram typically looks rather messy, so unless you spend some time in cleaning up the diagram, it doesn't make a lot of sense to version these files. They are usually created in a separate folder (/model by default), so you can exlude the whole folder from versioning.

When using TortoiseSVN, the "Add to Ignore List" option in Windows Explorer is only available on unversioned folders directly below a versioned folder. When committing a project for the first time, it is easier to exclude folders using the right-mouse-click popup menu in the Commit dialog, as shown in the screen shot below.


Enter Log Message - K:\adffaces\MyDemoSecure
Commit to:
https://adf-faces.ocsprojects.com/svn/adf-faces/trunk/www/adffaces/MyDemoSecure
Message:
Recent messages
Changes made (double-click on file for diff):
File
Extension
Text status
Property status
Lock
ViewController/model
Open
Open with...
Explore to
Delete
Add
Add to ignore list
Copy paths to clipboard
Copy all information to clipboard
ViewController/model/public_html/WEB-INF/JhsCommon-beans.oxd_faces
ViewController/model/public_html/WEB-INF/JhsCommon-beans.oxd_faces.bak
ViewController/model/public_html/WEB-INF/Jobs-beans.oxd_faces
ViewController/model/public_html/WEB-INF/Jobs-beans.oxd_faces.bak
ViewController/model/public_html/WEB-INF/region-metadata.oxd_faces
ViewController/model/public_html/WEB-INF/region-metadata.oxd_faces.bak
Show unversioned files
Select / deselect all
1 files selected, 64 files total
Keep locks
OK
Cancel
Help

2.2. Setting up Structure of JDeveloper Workspace and Projects

2.2.1. Installing JDeveloper

It is recommended that all members of the development team use the same version of JDeveloper. Different JDeveloper versions ship with different ADF libraries, which can lead to unexpected behavior when running the application using the Embedded OC4J container.

We also recommend that each developer installs JDeveloper in the same directory on their local PC. This is easy when you need to work/help on another PC, but more important, it prevents problems when you start using the facility to import Business Components from another project or jar file into your own project. When importing Business Components, JDeveloper stores path info of the imported components in the Model.jpx file so they can be displayed properly when using the ADF Business Component editors. If developers have a different JDeveloper directory, the path info might be incorrect and JDeveloper will not be able to find the imported business components.

Note that JHeadstart itself will import JHeadstart Runtime business components into your own Model project when you use one of the following features:

- Flex items or customizable standard items
- Custom Security
- Database table as resource bundle
- Dynamic menu structure

2.2.2. Identify Subsystems within your Application

It is good practice to organize your application into logical subsystems. These subsystems can be used as a basis for

- The projects you create within your JDeveloper workspace, see section Creating a Workspace and Projects.
- The java package structure (see section Defining Java Package Structure and Other Naming Conventions),
- The structure of your ADF Business Components. We recommend to create one ADF BC Application Module for each subsystem, that holds the View Object Usages needed to implement the business logic, and web pages for that subsystem.
- Dividing the work over the developers in the team.
- The structure of the JHeadstart Application Definition files. You will typically create one Application Definition for each subsystem, which can be based on the associated subsystem application module. See section Organizing JHeadstart Application Definition Files.

2.2.3. Creating a Workspace and Projects

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

- Create a New Application (choose menu option File – New – General – Application).
- 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 [JSF, ADF BC]** from the dropdown list. Although you can use JHeadstart in any kind of JDeveloper project, the recommended way is to use this application template as it is configured for building a data-bound web application.

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.

If you are building a large application, based on a database schema of say 100 or more tables, you might consider to create multiple Model projects.

Reasons to create multiple Model projects include:

- A layer of entity objects and/or view objects, and associated business rules will be used by multiple applications. In such a situation it makes sense to create a separate Model project for these entity objects (in a separate workspace if you like), create a Jar file of this model project which can then be imported into your application-specific Model project so you can create view objects on top of these entity objects. Note that the entity objects can only be changed in the owning Model project, not in the project in which they are imported.

ADF Developers Guide, section 25.7 "Working with Libraries of Reusable Components". Includes instructions on importing business components into another project.
http://download-west.oracle.com/docs/html/B25947_01/bcadvgen007.htm - CHEFECGD

- The application is very large and can be divided into subsystems with few dependencies on each other. Separate teams of developers work on each sub system. In this case it makes sense to split the subsystems into multiple model projects to have a clear separation of responsibilities, and to reduce the load time of the Model projects in JDeveloper. To handle the few dependencies between the subsystems, the facility to import business components can be use as described before.

We do not recommend to create multiple ViewController projects. The application needs to be deployed as one web application (.war file), and creating multiple ViewController projects will create duplicates of files that will complicate the deployment process.

If you are building a large web application, we recommend to use the option to define Working Sets on your ViewController project.

JDeveloper Online Help "Managing Working Sets". Includes instructions on creating working sets. Note that working sets are only supported in the System Navigator view, not in the Application Navigator.
http://www.oracle.com/webapps/online-help/jdeveloper/10.1.3/state?navSetId=_&navId=4&vtTopicFile=working_with_projects/ide_pworkingsetsmanaging.html&vtTopicId=

A working set allows you to define a set of files (a subset of project source path contents) that you want to work with, for example all files related to a subsystem. Typically, you would perform the following actions scoped to the working set:

- Make
- Build/rebuild
- Search
- Find usages

When you have defined some useful working sets, you can exchange them with your development team members. Here are the steps to do so:

- Open [JDevHome]/jdev/system/oracle.ide.10.1.3.xxxx/projects/index.xml and finding the entry for the JDeveloper project on which you defined the working sets.
- The entry in index.xml references a working set definition file in the [JDevHome]/jdev/system/oracle.ide.10.1.3.xxxx/projects folder.
- Possibly rename this (project-specific) working set definition file to a more meaningful name.
- Share this working set definition file with your team, and tell everyone to change the project entry in their own [JDevHome]/jdev/system/oracle.ide.10.1.3.xxxx/projects/index.xml to let it point to the new file.

2.2.4. Creating Database Connection

Create a Database Connection to the schema that contains the database tables of your application. Make sure that every developer uses the same name for the connection, and that they all make the same setting for the "Deploy Password" checkbox.

Suggestion: JDeveloper allows you to make an export of one or more database connections to an XML file. This XML file can then be used by other developers to import database connections, ensuring that the connections will be identical on all developer PC's. To use this feature, go to the Connections tab, right-mouse click on the Database node, and choose "Export Connections…"

2.2.5. Initializing Model Project for Business Components

Go to the Project Properties of the Model project, to the Business Components panel. Tick the checkbox 'Initialize Project for Business Components'. Choose the Database Connection you just created.

2.2.6. Optimizing ADF BC for Team Development

Go to the Model project, Project Properties. On the Business Components | General panel, uncheck the property named 'Copy Package XML Files to Class Path'.

This sets the default setting to be used for new ADF Business Components project. By unchecking this, the ADF design time no longer uses package XML files to track what components are in the package, so the package XML files will not be a point of merge conflicts between team members.

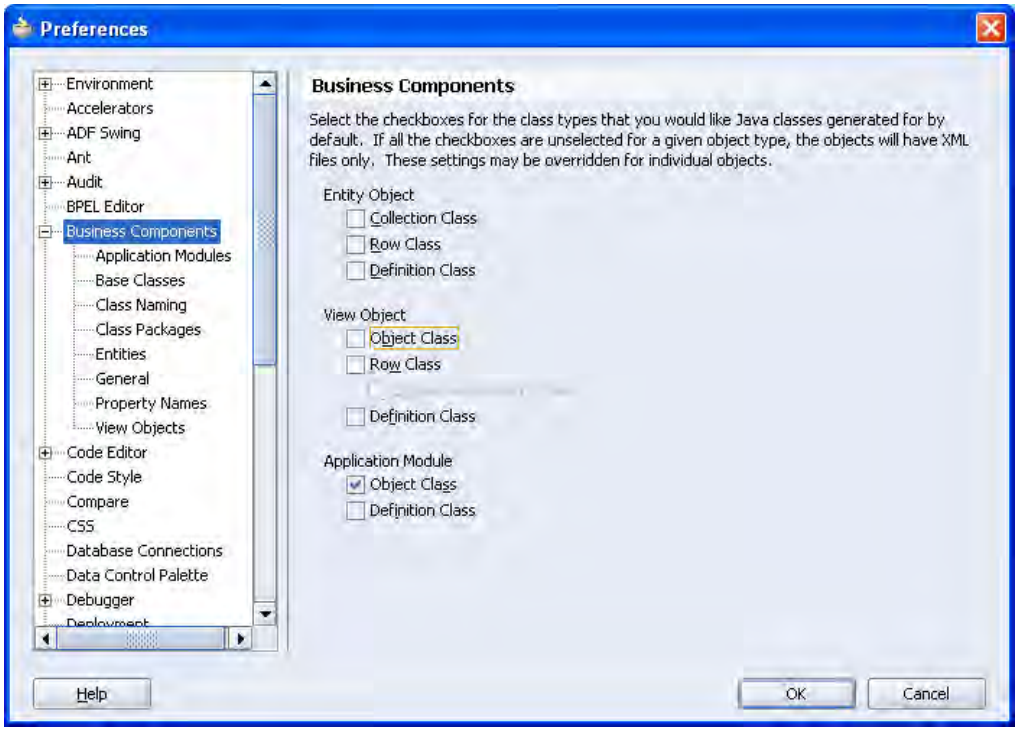
Suggestion: For existing projects, you can also edit this property at any time by unchecking it on the Business Components | Options panel of the project properties.

2.2.7. Switching off Default Creation of ADF BC Java classes

When creating entity objects, view objects and application modules, JDeveloper generates Java classes for all these components that you can use to add custom code. However, in most cases you will not add custom code to those classes, so it is better to turn off the creation of these classes since ADF Business Components does not require these classes to run your application.

If later on you do need to add custom code, you can still generate the Java class by going to the "Java" tab in the editor of the business component.

You can switch off the default creation of these classes by going to the Tools -> Preferences menu, and choose Business Components.



If you plan to implement business rules in ADF Business Components, you typically code these rules in the entity obbject row classes, so you could decide to create these classes upfront, and also check the Entity Object Row Class checkbox.

2.2.8. Enabling ViewController Project for JHeadstart

Before you can use JHeadstart in a project, you must first "Enable JHeadstart" on it. See chapter 4 "Using JHeadstart Addins" for more information.

2.3. Organizing JHeadstart Application Definition Files

As explained in chapter "Using JHeadstart Add-Ins", the JHeadstart Application Generator is driven the Application Definition file that holds the generator metadata. Although the name of the file suggests that you create only one application definition file for your entire application, we do recommned to create multiple application definition files, unless your application is really small. A typical approach is to create one application definition file for each logical subsystem. Since an application definition file is based on one data control (Application Module), the structure of your application definition files will typically follow the structure of your ADF BC application modules, which in turn should map your subsystem structure.

2.3.1. Service Level Settings that Should Be the Same Accross Application Definitions

When you create multiple application definition files, you should be aware of the fact that a number of settings you make at the service-level in the Application Definition Editor, are really application-wide settings. In other words, some service-level settings should be the same in all application definition file. Failing to keep these settings in synch might result in unexpected errors when running the application. For example, if the NLS resource bundle name property is different, then depending on the application definition file last generated, some pages might not be able to find the translatable strings, because they are in a different resource bundle, no longer configured for use in JhsCommonBeans.xml.

Here is a list with the service-level settings that should be the same across all application definition files:

- Generator Flavours: View Type and JSP Version
- File Locations: Common Beans Faces Config
- Java: View Package, Page Definitions SubPackage
- UI Settings: Date Format, DateTime Format
- Security: Authentication Type, Use Role-based Authorization, Authorization Type, Authorize Using Group Permissions
- Internationalization: NLS Resource Bundle, Resource Bundle Type, Generator Default Locale, Generator Locales, Read User Locale from, Generate Locale Switcher
- Runtime Customizations: Allow Runtime Customization of Menu, Allow use of Flex regions, Allow Runtime Customization of Items

2.3.2. Naming Conventions for File Location Properties

To cleanly organize the output produced by the JHeadstart Application Generator, it is helpful to set naming conventions for the File Location properties thet can be set at the service-level of an application definition file.

Here are suggested naming conventions.

Property	***Value***
Main Faces Config	*/WEB-INF/faces-config-<subsystem>.xml*

	For example: /WEB-INF/faces-config-hr.xml
Group Beans Faces Config Directory	*/<subsystem>/beandefs/*
UI Pages Directory	*/<subsystem>/pages/*
UI Page Regions Directory	*/<subsystem>/regions/*

By using a subsystem indication (short name or abbreviation) in the name, all files of a subsystem can easily be located. Since only one main faces-config is generated for each application definition (which only holds the navigation rules) , this file is not organized into a subsystem directory.

Note that by using this naming convention, JHeadstart will not generate the default faces-config.xml file, that is created and updated by ADF when using the visual design-time tools to create a JSF managed bean. So, with these settings you will never accidently wipe out custom managed bean definitions when running the JHeadstart Application Generator.

One drawback of not generating the default faces-config.xml is that ADF will add the default <lifecycle> element to the faces-config.xml when performing a drag and drop operation from the data control palette:

```
<lifecycle>
    <phase-listener>oracle.adf.controller.faces.lifecycle.ADFPhaseListener</phase-listener>
</lifecycle>
```

Since JHeadstart uses a customized version of the ADFPhaseListener (see chapter "JSF-ADF Page Lifecycle" for more info), you should remove this element again after your drag and drop action. If you forget to do this, you might loose the transactional messages displayed by JHeadstart when pressing Commit, or all messages might be displayed twice. Since you might easily forget to remove this element if you perform a lot of drag and drop actions on the generated pages, a more structural solution is to move the JHeadstart–required <lifecycle> definition from JhsCommon-beans.xml to faces-config.xml:

```
<lifecycle>
  <phase-listener>oracle.jheadstart.controller.jsf.lifecycle.JhsADFPhaseListener</phase-listener>
</lifecycle>
```

To prevent generation of this element into JhsCommon-beans.xml, you should create a custom template for JhsCommonBeans.vm and remove the element in this customized template.

2.4. Defining Java Package Structure and Other Naming Conventions

When working in a team, it is important to have standards on naming java packages and the various types of (business) components. Everybody seems to have different opinions on naming conventions, but remember, the important thing is to have naming standards in place and have them applied by all developers. The actual format of the naming conventions is less important.

As a suggestion, here are the naming conventions as applied on projects by the JHeadstart Team, use them to your own advantage.

2.4.1. Java Packages

The root package of an application is by convention the reverse of your companies internet domain name, followed by the application name, for example “com.acme.hr”.

A suggested package structure within the application can be found in the table below. Subsitute the three dots with your application root package.

Package	Description
….model	Base package forbusiness service classes, classes who contain logic not specific for the web application built on top of it
….model.adfbc	Base package for ADF Business Components.
….model.adfbc.base	Package for base classes extended by the ADF Business Components you create for the application
….model.adfbc.entity	Base package for entity objects and associations
….model.adfbc.entity.<subsystem>	For larger applications, it is good practice to further organize entity objects into subsystem packages, for example “model.adfbc.entity.authorization”.
….model.adfbc.query	Base package for view objects and view links
….model.adfbc.query.<subsystem>	For larger applications, it is good practice to further organize view objects into subsystem packages, for example “model.adfbc.query.authorization”.
….model.adfbc.service	Contains application modules
….controller	Base package for classes that contain logic to control the behavior of the web application.
….controller.jsf	Base package for JSF-specific classes that contain logic to control the behavior of the web

	application.
….controller.jsf.bean	Contains JSF managed bean classes
….controller.jsf.lifecycle	Contains custom JSF Page Lifecycle classes
….view	Base package for classes that contain logic to display web pages and the user interface in general
….view.pagedefs	Contains ADF Model Page Definitions

2.4.2. Naming ADF Business Components

- Entity Object names are self-descriptive and in singular. Remove any table name prefixes from the name.

  example: Department

- Entity Associations are named using the format <master entity><verb describing relationship><detail entity>.

  example: DepartmentHasEmployees

- View Objects names describe the result of the query, are in singular when the query returns onew row at most, and in plural when the query can return multiple rows. Any bind parameters defined for the ViewObject, should be resembled in the name

  examples: AllClerks, ClerksByDepartment

- View Links are named using the format <master view object><verb describing relationship><detail view object>.

  example: DepartmentHasEmployees

- Application Modules names are descriptive for the functional area they cover, and are suffixed with “Service”.

  examples: AuthorizationService, HumanResourcesServic

CHAPTER

3

Creating ADF Business Components

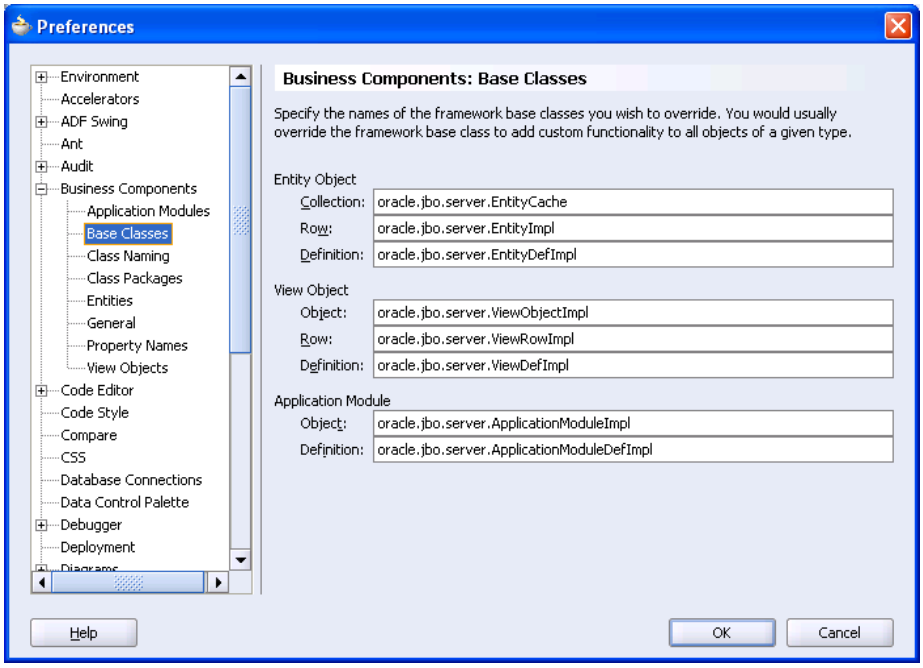
This chapter provides you with guidelines on creating ADF Business Components. The Oracle ADF Developer's Guide for Forms/4GL Developers Release 10.1.3 already contains a wealth of information about how to use ADF Business Components. This chapter will not duplicate that information. It focuses on best practices collected by Oracle Consulting and guidelines on how you can best set up and prepare your ADF Business Components when using JHeadstart to generate the View and Controller layers.

If you are new to ADF Business Components, we strongly recommend to first read chapters 4 to 9 of the ADF Developer's Guide.

Oracle ADF Developer's Guide for Forms/4GL Developers, Chapters 4 to 9 describe the basics for creating ADF Business Components. Chapters 25 to 28 contain advanced techniques.
http://www.oracle.com/technology/documentation/jdev/B25947_01/index.html.

3.1. Setting Up ADF BC Base Classes

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 ADF Developer's Guide 10.1.3 it is recommended to create your own layer of ADF BC Base Classes, also called framework extensions:

Before you begin to develop application-specific business components, Oracle recommends that you consider creating a complete layer of framework extension classes and setting up your project-level preferences to use that layer by default. You might not have any custom code in mind to put in some (or any!) of these framework extension classes yet, but the first time you encounter a need to:

- *Add a generic feature that all your company's application modules require*
- *Augment a built-in feature with some custom, generic processing*
- *Workaround a bug you encounter in a generic way*

You will be glad you heeded this recommendation. Failure to set up these preferences at the outset can present your team with a substantial inconvenience if you discover mid-project that all of your entity objects, for example, require a new generic feature, augmented built-in feature, or a generic bug workaround. Putting a complete layer of framework classes in place to be automatically used by JDeveloper at the start of your project is an insurance policy against this inconvenience and the wasted time related to dealing with it later in the project.

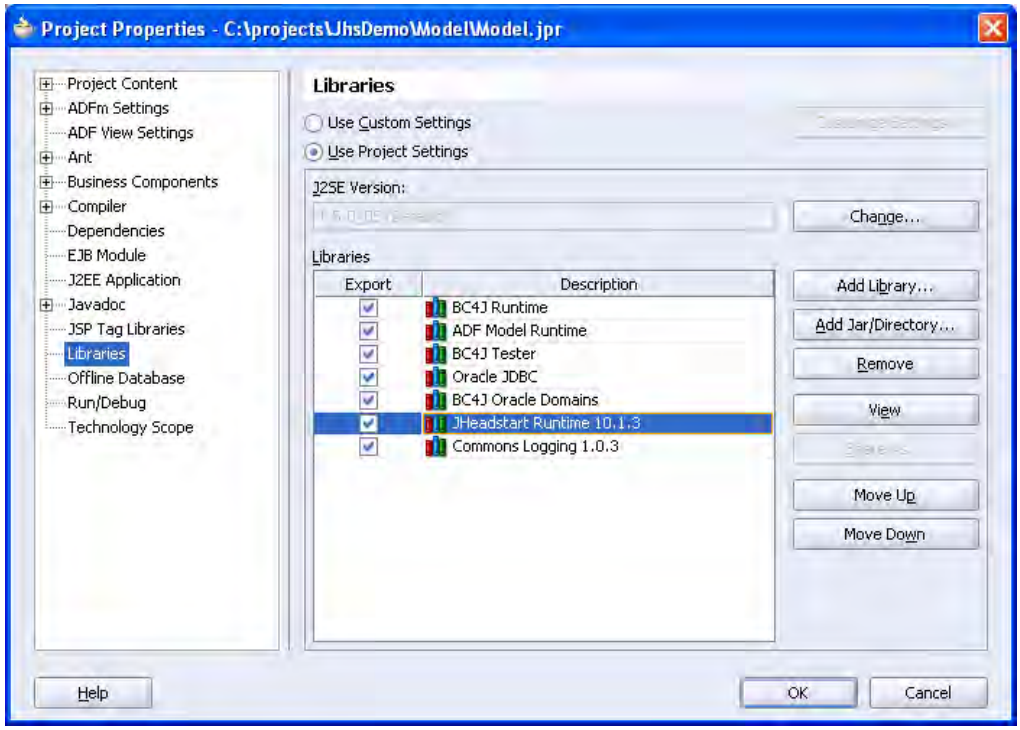
For an explanation how to create such a layer, see the ADF Developer's Guide.

Reference: See the Oracle Application Developer Framework Developer's Guide for Forms/4GL Developers Release 10.1.3, chapter 25: *Advanced Business Components Techniques*, section 25.1. *Globally Extending ADF Business Components Functionality*, and section 25.2. *Creating a Layer of Framework Extensions*. Internet: http://download-uk.oracle.com/docs/html/B25947_01/bcadvgen.htm#sm0291.

The Application Module Object base class can be used to implement functionality that is needed in all Application Modules of your application. Therefore JHeadstart has created is own subclass of the standard `oracle.jbo.server.ApplicationModuleImpl` class. (The JHeadstart Application Generator can automatically set up the use of this class if you don't have your own ADF BC framework extensions.)

If you want additional custom functionality for your application modules, this means that your custom AppModuleImpl should not extend the standard base class but rather the JHeadstart base class**: `JhsApplicationModuleImpl`**.

To do this, you must first make sure that the JHeadstart Runtime 10.1.3 library is added to your project. Also add the Commons Logging library, because JHeadstart uses it.

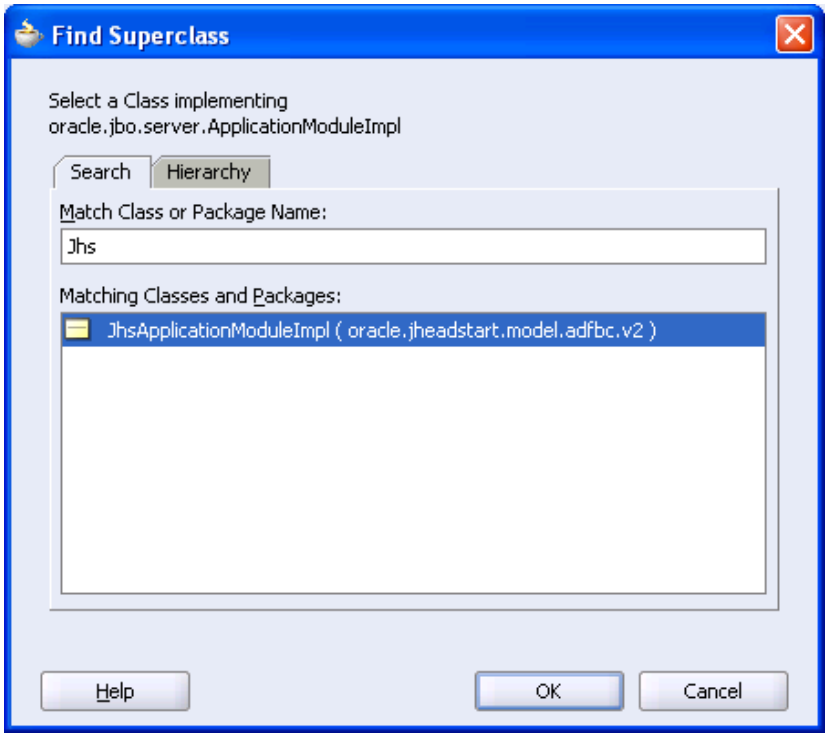


- Go to the Project Properties
- Select category Libraries
- Click Add Library
- Under Extension, select Commons Logging 1.0.3 and Ctrl-click to also select JHeadstart Runtime 10.1.3
- Click OK twice

- Save the project

Now you can use `oracle.jheadstart.model.adfbc.v2.JhsApplicationModuleImpl` as the super class of your custom Application Module framework extension.

Suggestion: In the Base Classes wizard page, use the Browse button to find the desired base class. In the Search field, type in the first letters of the class name, and the dialog will show all available classes that satisfy the base class requirements.



3.1.1. Using CDM RuleFrame

CDM RuleFrame is a PL/SQL based framework for implementing business rules in the database, tightly integrated with Oracle Designer.

Headstart Oracle Designer. Add on to Oracle Designer that includes CDM RuleFrame:
http://www.oracle.com/technology/products/headstart/index.html

If you use CDM RuleFrame to implement business rules, you want the errors reported by CDM RuleFrame to be displayed nicely in your generated web application. Using JHeadstart this is easily accomplished by using a special application module super class shipped with JHeadstart: `RuleFrameApplicationModuleImpl`. So, when using CDM RuleFrame, your application module base class should extend `RuleFrameApplicationModuleImpl` rather than `JhsApplicationModuleImpl.`

Note that `RuleFrameApplicationModuleImpl` extends `JhsApplicationModuleImpl` in turn; so all standard JHeadstart functionality is still available.

3.2. Creating the Entity Object Layer

This section discusses the development tasks related to creating the entity object layer. The following topics are discussed:

- Review Database Design
- Creating First-Cut Entity Objects and Associations
- Renaming Entity Objects and Associations
- Generating Primary Key Values
- Setting Entity Object Attribute Properties used by JHeadstart
- Implementing Business Rules

3.2.1. Review Database Design

A sound database design is critical to successfully building a performant ADF Business Components layer. Providing guidelines for sound relational database design is outside the scope of this developer's guide, however, some guidelines directly impacting the behavior of your web application are discussed below:

- If you are in the position to create or modify the database design, make sure all tables have a non-updateable primary key, preferably consisting of only one column. If you have updateable and/or composite primary keys, introduce a surrogate primary key by adding an ID column that is automatically populated. See section 3.2.4 Generating Primary Key Values for more info. Although ADF Business Components can handle composite and updateable primary keys, this will cause problems in ADF Faces pages. For example, an ADF Faces table manages its rows using the key of the underlying row. If this key changes, unexpected behavior can occur in your ADF Faces page. In addition, if you want to provide a drop down list on a lookup tables to populate a foreign key, the foreign key can only consists of one column, which in turn means the referenced table must have a single primary key column.
- Ensure that all the primary key, unique key, and foreign key constraints and check constraints that logically exist, are explicitly defined as database constraints in your database server. When you create ADF Business Components, these database constraints are stored in the Entity Object XML file. JHeadstart uses this constraint information to generate user-friendly error messages when a database constraint is violated.

3.2.2. Creating First-Cut Entity Objects and Associations

Use the JDeveloper wizard *Business Components from Tables* to create entity objects for your database tables. You can find this wizard in the New Gallery. On the "Create Entity Objects" wizard page, press the Query button to see all tables you created in the previous exercise.

Make sure you specify a proper package name for the entity objects, based on the naming conventions that you specified for your project. See chapter 2 for more info on naming conventions.

Do **not** yet create default updateable or read-only view objects and do **not** create an application module using the wizard for two reasons:

- We first rename the entities and associations and the new names will be used when we create view objects and view links.
- A default application module typically contains many (nested) view object usages that you do not need in your application. You should set up your application module data model based on the layout of the pages you will create to meet the functional requirements of your application.

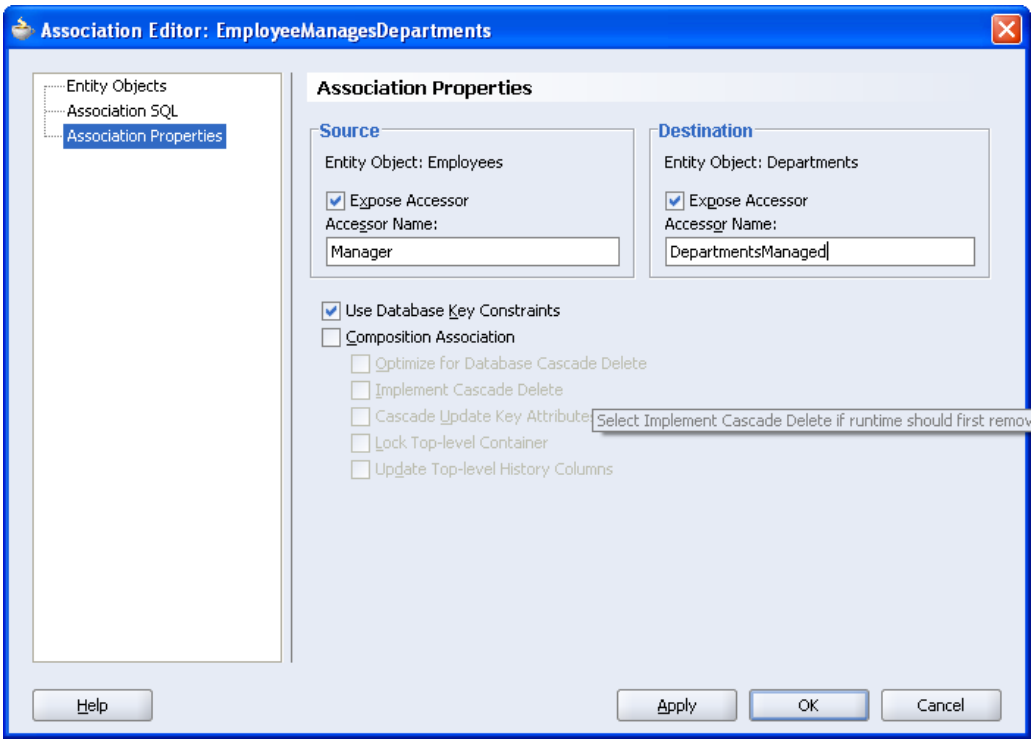
3.2.3. Renaming Entity Objects and Associations

We recommend renaming entity objects and associations to comply with the naming standards you have set up for your project (see chapter 2).

Renaming associations and the association accessors is important for the following reasons:

- By default, associations are named after the foreign key constraint suffixed with "FkAssoc". Foreign key names are often not very meaningful, so if you have many associations, they are easier to manage with meaningful names.
- The accessor properties of an association determine the accessor method names generated into the entity objects to traverse entity object associations, something you will often do when coding business rules in your entity objects. Logical accessor methods make it easier to code this logic.
- The view links that are created when you create default view objects for your entity objects are named after the underlying associations, saving you an additional renaming of the view links (although you typically will remove the "Link" suffix added to the view link name).

For example, when you create entity objects for the EMPLOYEES and DEPARTMENTS tables in the HR schema of the Oracle database, an association named DeptMgrFkAssoc is created, with association properties named "Employees" and "Departments". We recommend renaming the association to something like "EmployeeManagesDepartments", and the association properties to "Manager" and "DepartmentsManaged".



Now, if you need to code logic in the Employee Entity Object that requires access to the departments an employee manages, you can call the `getDepartmentsManaged()` method rather than the confusing `getEmployees()` method.

3.2.4. Generating 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.

3.2.4.1. 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'.

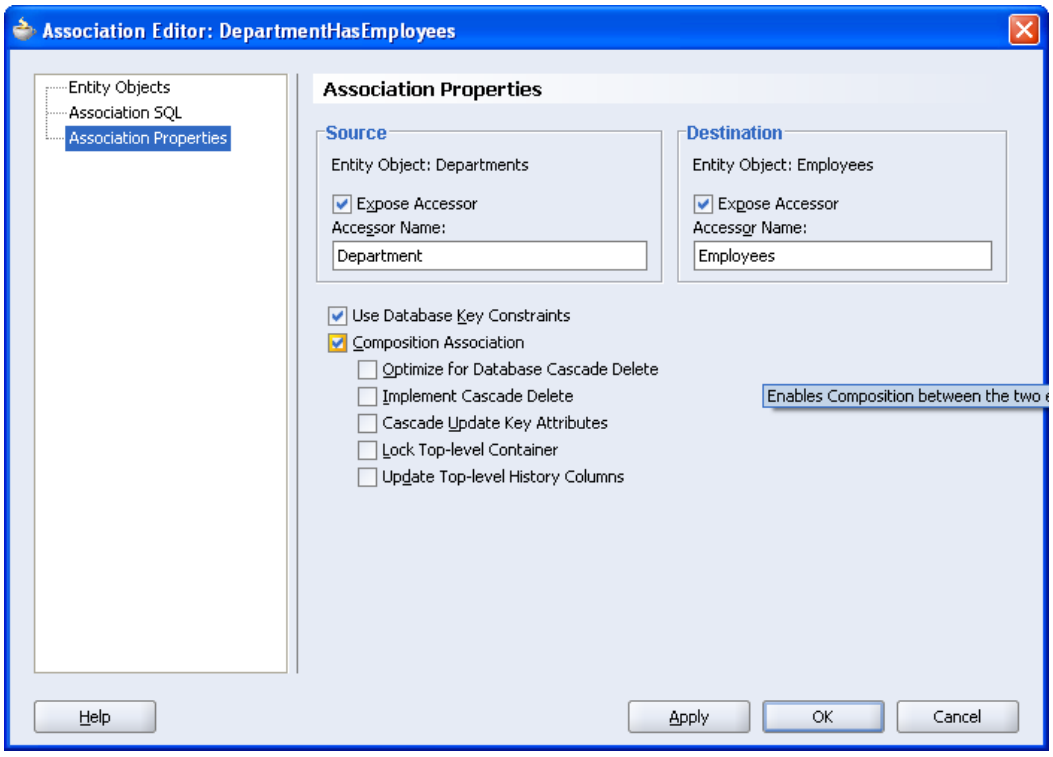
Suggestion: If all primary key attributes have the same name, for example Id, retrieving sequence values from the database in the create method is something you could implement in your BC base classes. By doing so, you do not need to implement a create method for each entity object. In the EO base class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

3.2.4.2. 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.

ADF Developer's Guide, section 6.6.3.8 "Trigger-Assigned Primary Key Values".
http://download-west.oracle.com/docs/html/B25947_01/bcentities006.htm#sm0147

When you plan to create screens that insert a master row and one or more detail rows in one transaction, you will need to ensure that ADF BC first posts the master row and then the detail rows, otherwise ADF BC will not be able to set the foreign key of the details rows correctly. To enforce this posting sequence, you should mark the entity association as "Composite Association".



Note that when you mark an association as composite, the detail entity object can only be created as a detail of the master entity object, which means you cannot create a page that directly creates detail entity object, in addition to the master-detail page. If you try to do so, you will get error **JBO-25030: Failed to find or invalidate owning entity.**

Weblog Steve Muench: "Why do I get InvaliOwnerException".
http://radio.weblogs.com/0118231/stories/2003/01/17/whyDoIGetTheInvalidownerexception.html

3.2.5. Setting Entity Object Attribute Properties used by JHeadstart

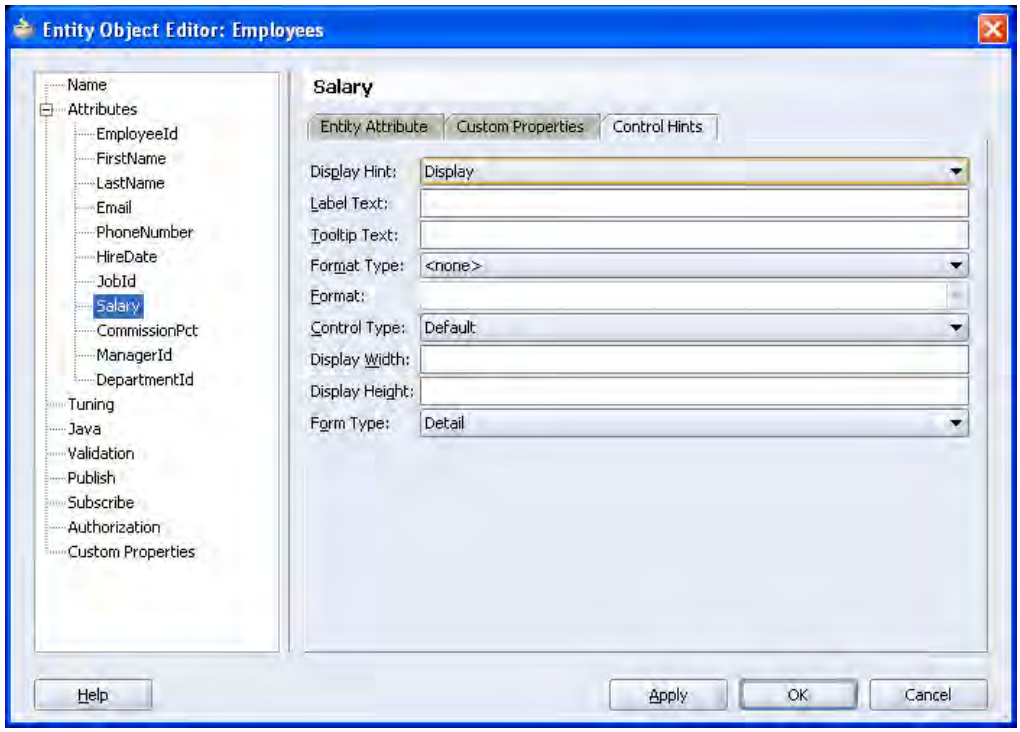
A number of properties that you can set on the Entity Object attribute panel are carried forward by JHeadstart into the Application Definition file used to generate the application:

- **Mandatory**: when this checkbox is checked, the item that is based on this attribute will be generated with an asterisk in front of the label, and a JavaScript alert will be displayed when you submit the page when the item is still empty.
- **Queryable**: when checked, the item created for this attribute in the Application Definition will have the **Show in Quick Search** and **Show in Advance Search** checkboxes checked by default. Of course, you can uncheck these checkboxes later on.
- **Updateable**: when set to "While New" the item will be generated as a read-only item when an existing row is displayed on the page. When set to "Never" the item will be read-only in the generated page.

Note that Queryable and Updateable properties can also be defined at the view object (VO) level. An item that is queryable at EO level can be made non-queryable at VO level. An item that is updateable at the EO level can be made (partly) read-only at the VO level. The VO level settings take precedence when JHeadstart creates the Application Definition file.

3.2.5.1. Specifying Entity Object Control Hints

You can specify Control Hints for an attribute in an Entity Object. See the screenshot below.



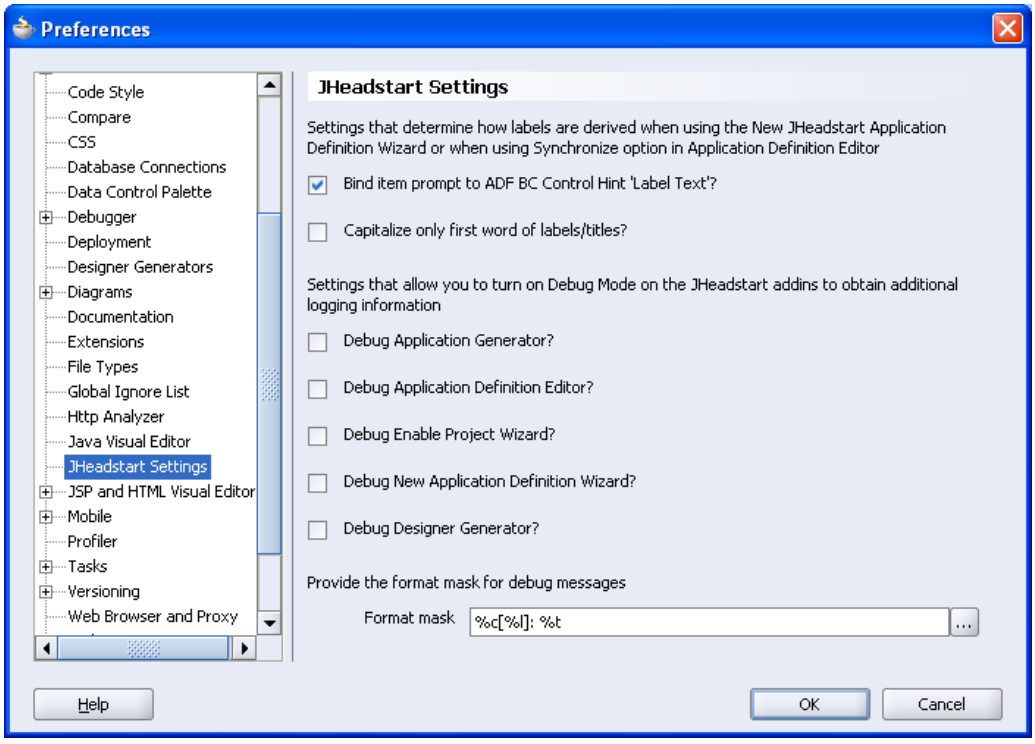
The information you enter in this panel is stored in a resource bundle for the entity object. This means that if you want to translate your application, you have to go through numerous resource bundles, created for all your entity and view objects (which also support control hints) and make language-specific copies of all those bundles.

Internationalizing Control Hints. Explained in ADF Developers Guide. http://download.oracle.com/docs/html/B25947_01/bcentities005.htm#sm0140

This can be a tedious job when you have many ADF Business Components. When using JHeadstart, you have an easier and faster alternative.

By default, JHeadstart will *copy* the values for **Label Text**, and **Tooltip Text** to the corresponding item properties (**Prompt in Form Layout** and **Hint Text**) in the JHeadstart Application Definition file. Then, when you generate your application with the service-level property **Generate NLS-enabled prompts and tabs** checked, all translatable strings, including labels and tooltip/hint texts, are stored in one centralized resource bundle, or database table. One centralized bundle is much easier to manage and translate, and when you use the database table as the store for translatable strings, JHeadstart even offers an in-page editor to translate your application. See chapter 11 "Internationalization" for more information.

Now, if for whatever reason you want to use the business component resource bundles for translating labels and tool tips, instead of the JHeadstart resource bundle or database table, then you can do so by switching a JHeadstart preference. Go to the JDeveloper Tools menu and choose "Preferences…" At the left side click " JHeadstart Settings" and the panel displayed below appears.



If you check the checkbox "Bind Item prompt to ADF BC Control Hint "Label Text", JHeadstart will no longer copy the Label value over to the Application Definition. Instead, it will set the Label property of the item to an EL expression that references the label as defined in the Control Hints panel.

3.2.6. Implementing Business Rules

ADF Business Components has extensive support for implementing business rules, both declaratively using so-called validators, as well programmatically in the Entity Object implementation classes. The JHeadstart Team has written a comprehensive white paper on implementing business rules in ADF Business Components. This white paper can be downloaded from OTN, and includes a classification of business rules, and a structured approach to implementing each type of business rule.

Implementing Business Rules in ADF BC. White paper on OTN.
http://www.oracle.com/technology/products/jdev/collateral/papers/10131/businessrulesinadfbctechnicalwp.pdf

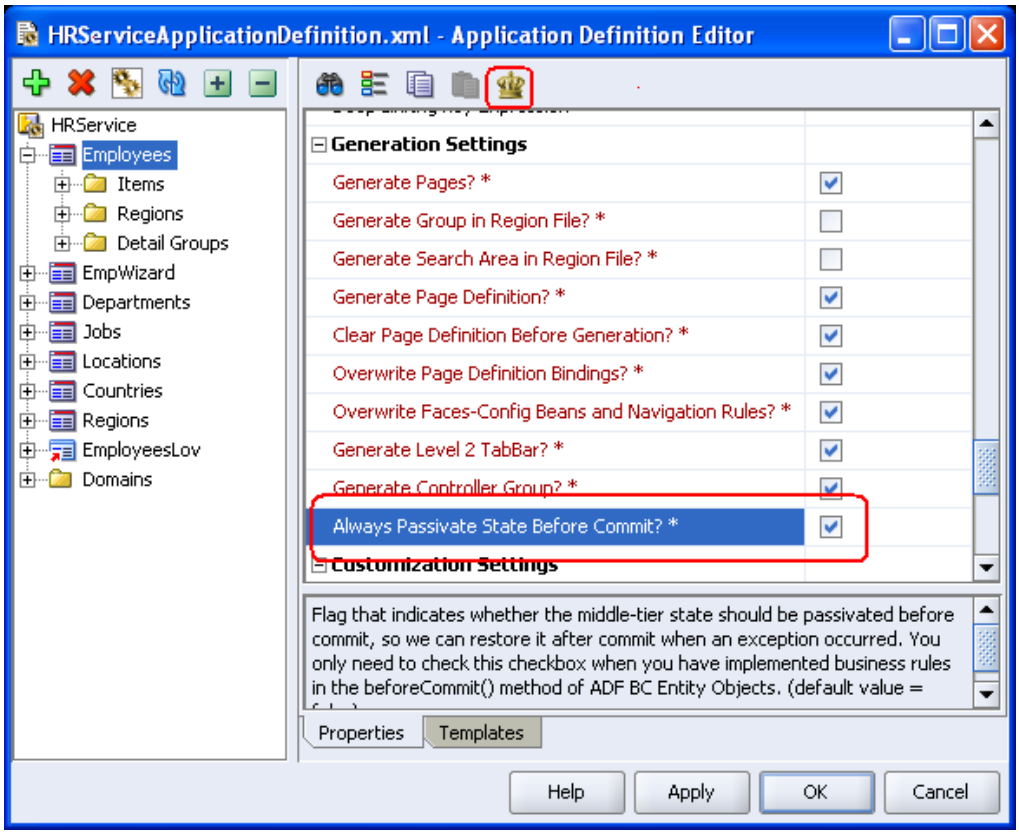
3.2.6.1. Adding Business Rules to the beforeCommit() method

If your business rule logic in the entity object `beforeCommit()` method can throw an exception you must set the `jbo.txn.handleafterpostexc` property to true in your application module configuration. By doing so the framework automatically handles rolling back the in memory state of the other entity objects that may have already successfully posted to the database (but not yet been committed) during the current commit cycle.

However, there is a known issue (bug 4606787) which causes the `jbo.txn.handleafterpostexc = true` setting to conflict with custom activation/passivations of the ADF Business Components state. When such a conflict occurs, the following exception will be thrown: **JBO-25033: Activating state from Database at id xx failed**.

JHeadstart uses custom activation/passivation in the `JhsPageLifecycle` class to restore rows already removed from ADF Business Components when the final database commit fails because of referential integrity constraint violations. This means you cannot set `jbo.txn.handleafterpostexc` to true, and if you nevertheless do so, you will get the JBO-25033 error.

Fortunately, JHeadstart provides functionality to work around this bug. Instead of setting the `jbo.txn.handleafterpostexc` property to true, you need to check the group-level checkbox **Always Passivate State Before Commit?** in the Application Definition Editor. Note that this setting is only visible in expert mode.



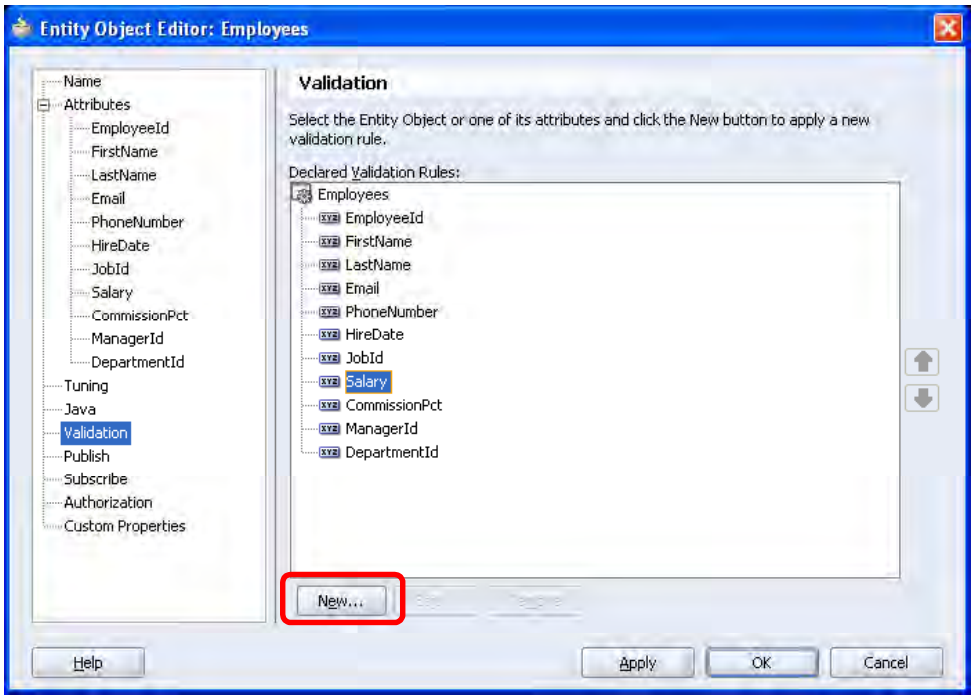
You need to make this setting for each group, which generates pages that might start transactions, which cause the beforeCommit () method to fire.

3.2.6.2. Define List Validators for Static Lookups

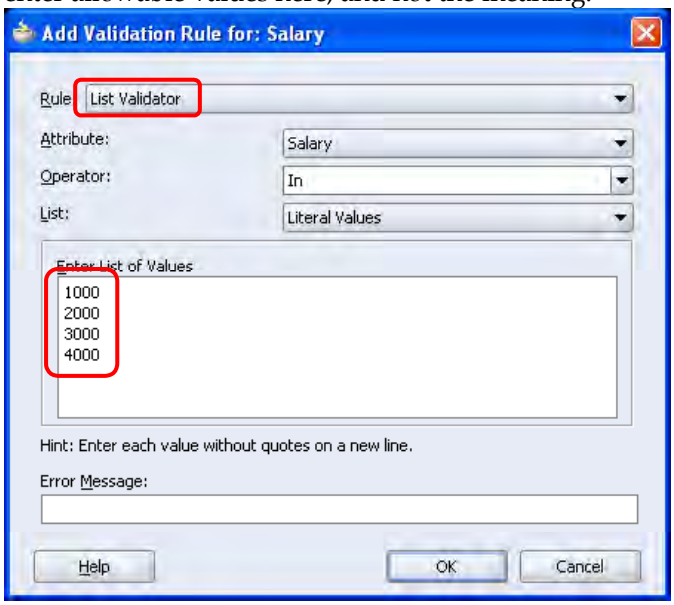
One topic, also discussed in this white paper, provides additional functionality in combination with JHeadstart: defining List Validators. ADF BC has the possibility to add Validators to Entity Objects. Using the List Validator, you can check for allowable values in the Model layer.

In this example we use a List Validator to check the allowable values for the salary column.

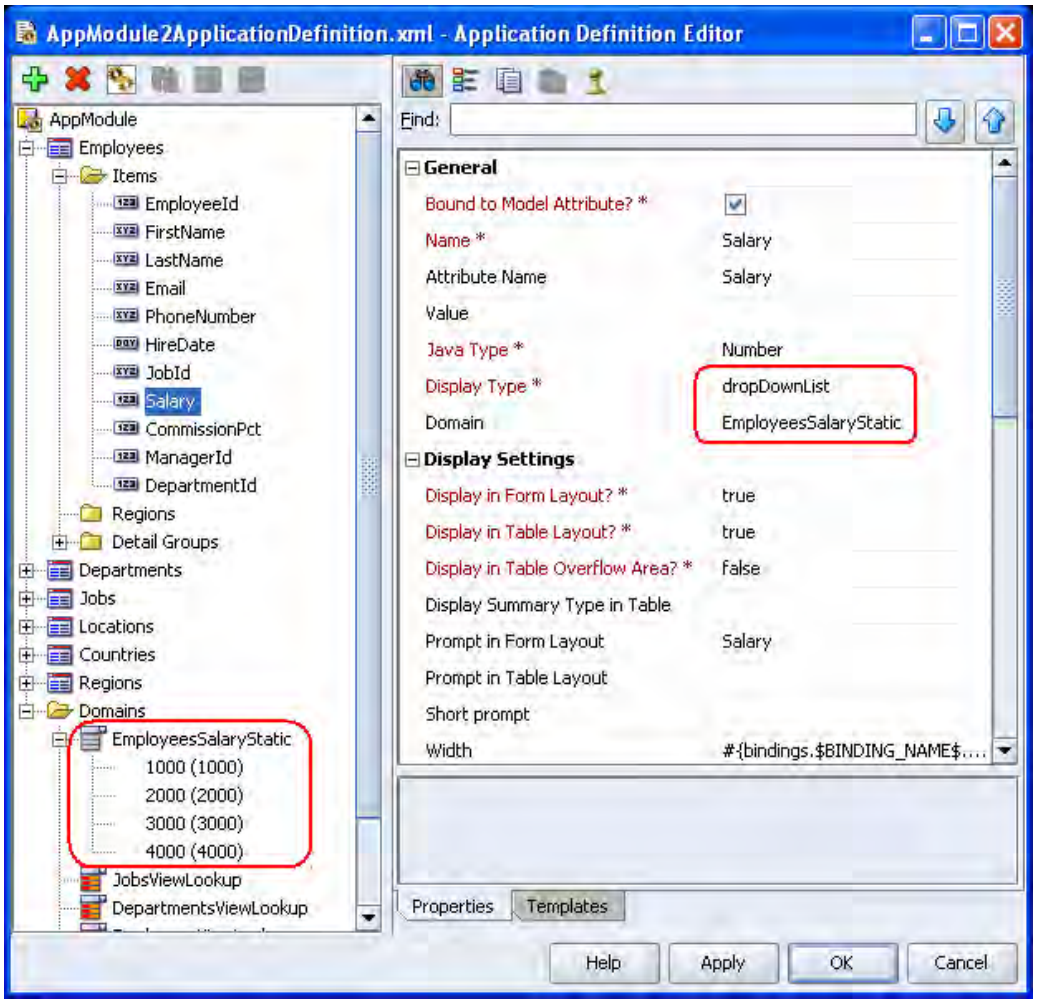
1. Edit the Entity Object and go to the Validation Node. Select the attribute you want the Validator for and press New:



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



Now, when you create a default Application Definition using the New JHeadstart Application Definition wizard, JHeadstart creates a static domain with the same set of allowable values as defined in the List Validator.



If you now generate the application, you will get a drop down list on salary with the allowable values as entered in the List Validator. If you prefer to have a radio group, you can change the Display Type for the item to either “radio-horizontal” or “radio-vertical”.

3.3. Creating View Objects and Application Modules

This section discusses the development tasks related to creating the data model layer, consisting of View Objects, View Links and Application Modules. The following topics are discussed:

- Creating View Objects and View Links
- Renaming View Objects and View Links
- Inspecting and Setting Key Attributes of a View Object
- Setting View Object Control Hints
- Determining the Order of Displayed Rows
- Creating Calculated or Transient Attributes
- Setting up Master-Detail Synchronization
- Defining View Links and View Object Usages for Lookups
- Testing the Model

3.3.1. Creating View Objects and View Links

When creating a View Object you need to determine whether the data queried through the ViewObject should be updateable in the user interface (web pages). If so, you need to create an updateable ViewObject, which is based on a primary Entity Object. If the data is read-only in the user interface, it is more efficient to create a read-only View Object, which is a View Object not based on an entity object with a custom SQL query that you need to enter manually.

A typical example of read-only View Objects, are View Objects used to populate lookup data in the user interface, typically exposed through a drop down list, or List of Values window.

3.3.2. Renaming View Objects and View Links

If you have used the "New Default Data Model Components" wizard, we recommend that you rename the View Objects and View Links to comply with the naming standards you have set up for your project (see chapter 2). If you create the View Objects and View Links one-by-one, you can assign proper names right away.

3.3.3. Inspecting and Setting Key Attributes of a View Object

Under the covers, an ADF Faces table uses the View Object `findByKey()` method for its row management. This row management is used by the ADF Faces table to update the correct underlying row, when a user has changed one or more values in the ADF Faces table. Built-in ADF Data Binding layer actions like setCurrentRowWithKey and setCurrentRowWithKeyValue also rely on the findByKey() method. For this method to behave reliably, two conditions must be met:

- Each View Object must have at least one key attribute
- The key attribute(s) should be non-updateable

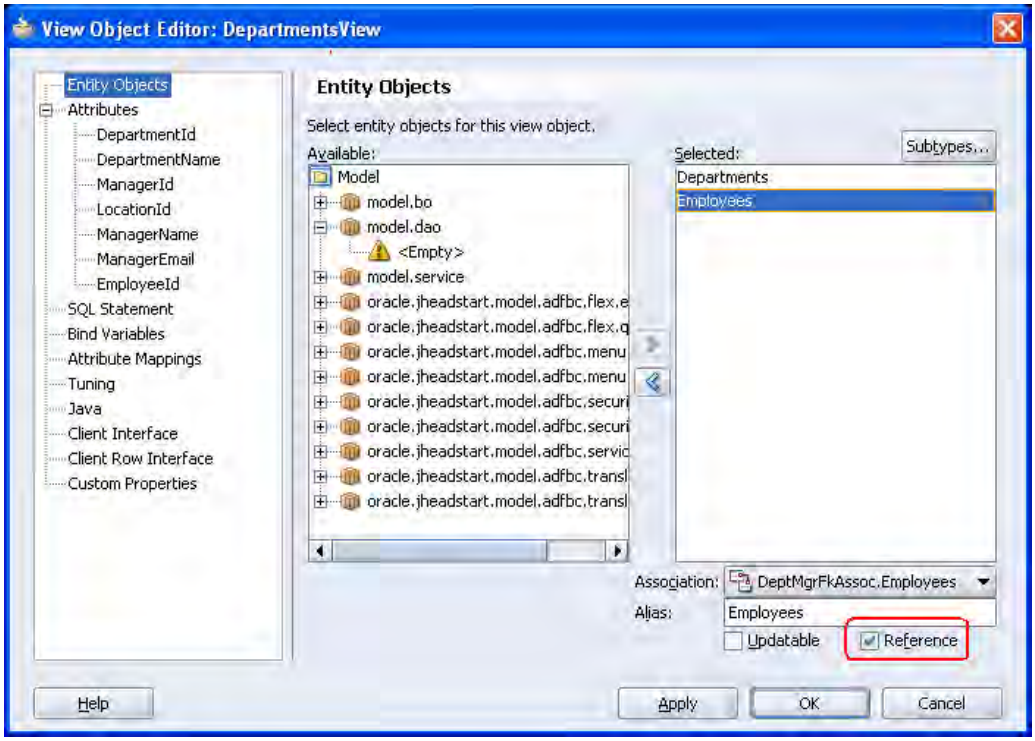
A view Object key that is updateable might result in unexpected behavior in the web tier. For example, if you update key attribute values in an ADF Faces table, the row management feature might not work correctly anymore.

To ensure the above two conditions are met; you must perform different tasks, depending on whether the View Object is Updateable or Read-Only.

3.3.3.1. Unchecking Reference Key Attributes for Updateable View Objects

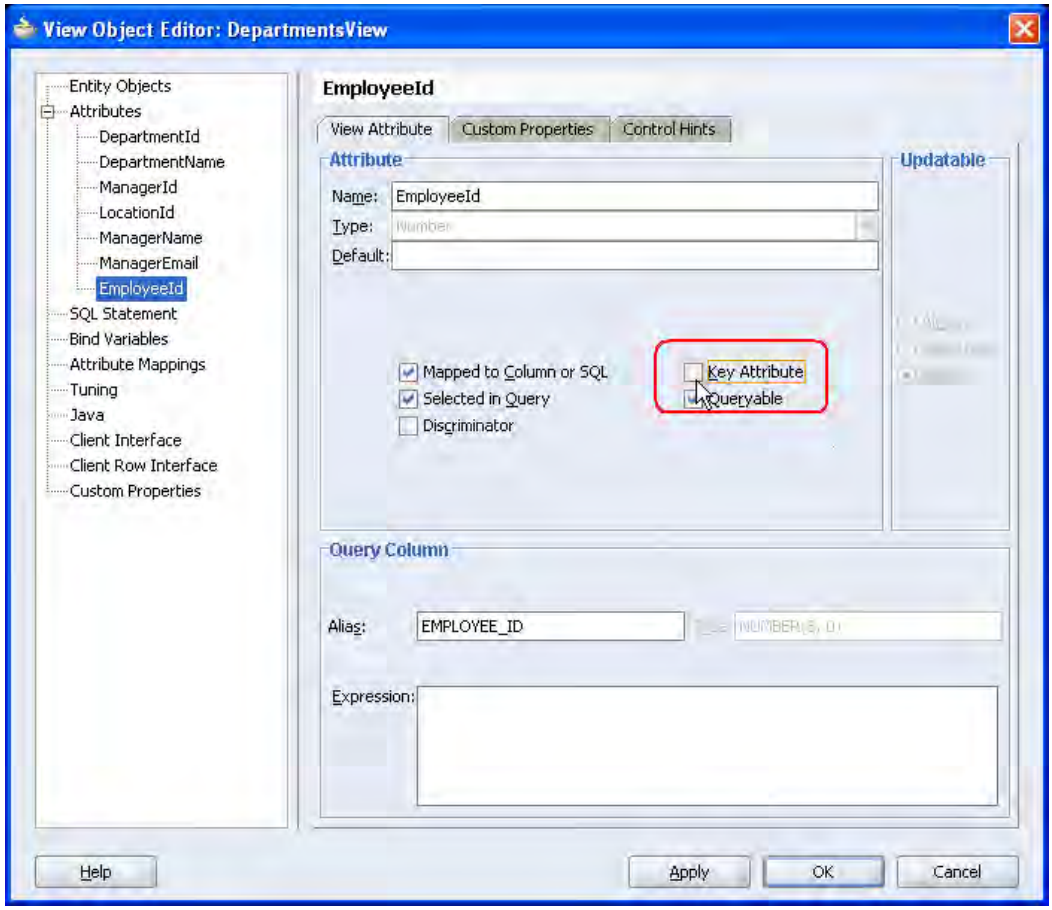
An updateble View Object based on only one Entity Object, will inherit its key attribute(s) from the underlying Entity Object, which in turn is inherited from the underlying table's primary key. Assuming you have applied common sense database design guidelines, the primary key is not updateable, hence the View Object key is read only as well. So far so good, however, when you start adding "Reference" Entity Objects to the View Object, for example to join lookup data in the query, ADF Business Components will add the key attributes of the reference Entity Object to the key of the View Object (JDeveloper bug 6804062, fixed in 11g).

For example, we want to display the Department Manager Name on a page that lists all departments. We need to join the Departments table with the Employees table in the View Object query. The easiest way to do this is by adding the Employees Entity Object as Reference Entity Object, as shown in the picture below.



Once you added the Employees Entity Object, and added some Employee attributes to the View Object, ADF Business Component nicely changes the SQL statement for you to join with the Employees table. However, the key attribute of the Employees Entity Object, EmployeeId is also checked, effectively adding this attribute to the Key of a View Object Row. So, ADF Business Components silently made your Row Key updateable. When you subsequently use JHeadstart to generate a List of Values window to change the manager of a department in a table layout, you will notice that the LOV values are not returned as expected. This is because the Row Key has changed in the middle of this

user action. The solution is shown below: uncheck the Key attribute checkbox of the "reference" key attribute(s), EmployeeId in this case.



3.3.3.2. Set Manage Rows By Key for Read-Only View Objects

When you create a read-only View Object, by default none of the attributes will be marked as Key attribute. In order to successfully be able to use the findByKey() method on a read-only view object, you need to perform two additional steps:

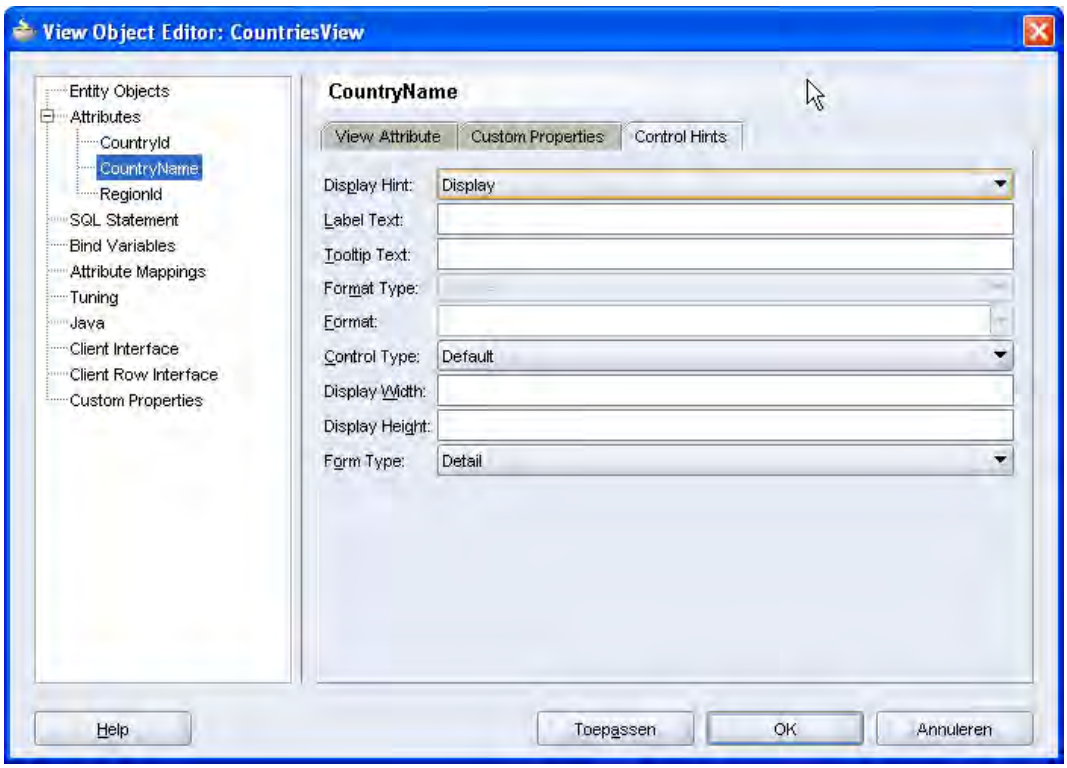
1. Ensure that at least one attribute in the view object has the Key Attribute property set, and make sure this is a non-updateable attribute.
2. Enable a custom Java class for the view object, and override its create() method to call setManageRowsByKey(true) after calling super.create() like this:

```
// In custom Java class for read-only view object
public void create()
{
  super.create();
  setManageRowsByKey(true);
}
```

Suggestion: Rather than adding this create() method to each and every read-only View Object, you can apply a generic coding technique in the View Object base class. See section 25.3.2 of the ADF Developer's Guide: http://download-uk.oracle.com/docs/html/B25947_01/bcadvgen003.htm - CHECHIJG

3.3.4. Setting View Object Control Hints

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

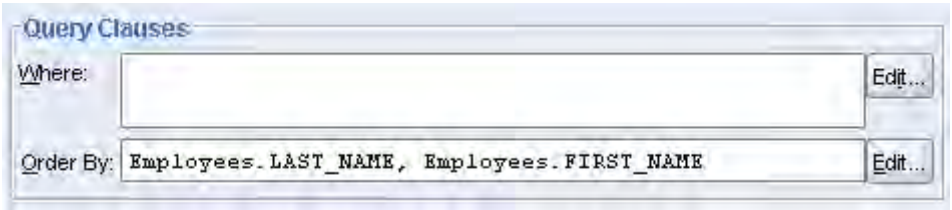


JHeadstart might use some of these properties in the same way as Control Hints specified for an Entity Object. See section Setting Entity Object Attribute Properties used by JHeadstart fore more information.

3.3.5. Determining 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 chapter 5, section "Allowing the user to sort data in a table page" for more detail on how to do this.

3.3.6. Creating 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.

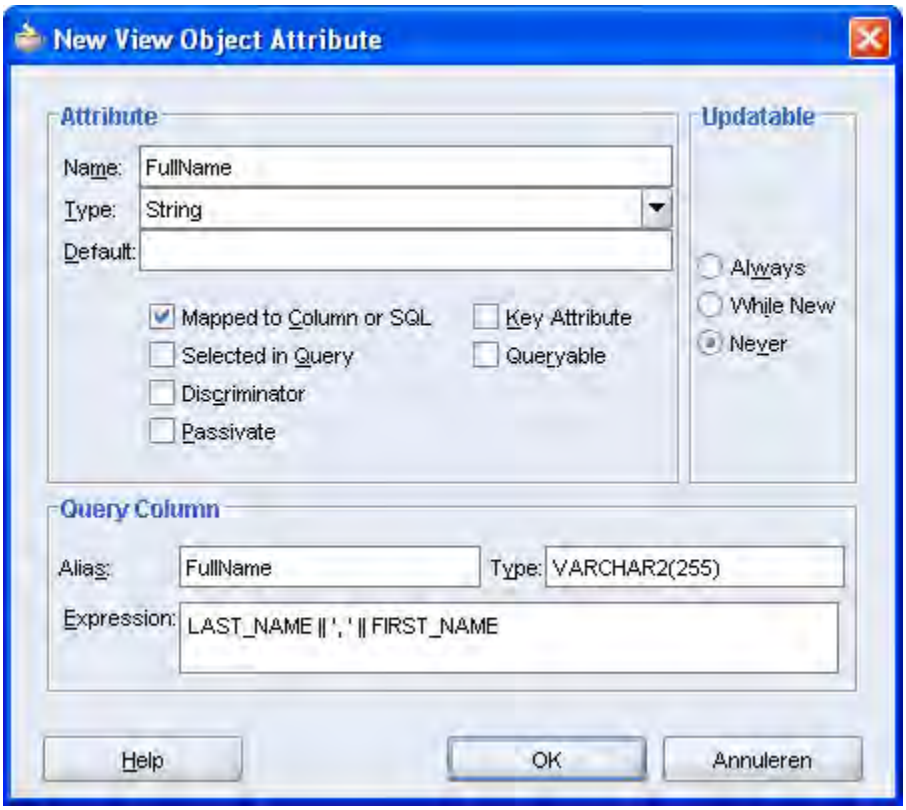
Note the important difference between a calculated and a transient attribute:

- 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 re-executed. 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 read-only 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.

Reference: See also the Oracle Application Developer Framework Developer's Guide for Forms/4GL Developers Release 10.1.3, section 7.6: Adding Calculated and Transient Attributes to an Entity-Based View Object.

3.3.6.1. 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 'Mapped to Column or SQL'.
5. Give the attribute an alias.
6. In the Expression text area, 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: LAST_NAME || ', ' || FIRST_NAME

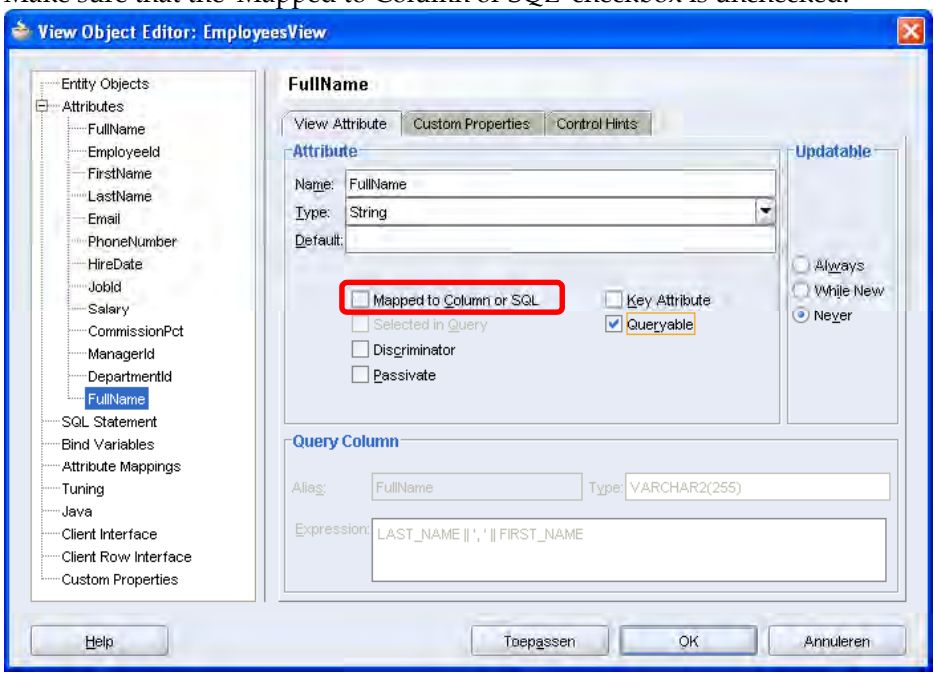


7. Save the changes and close the View Object dialog.

3.3.6.2. Steps to create a transient attribute

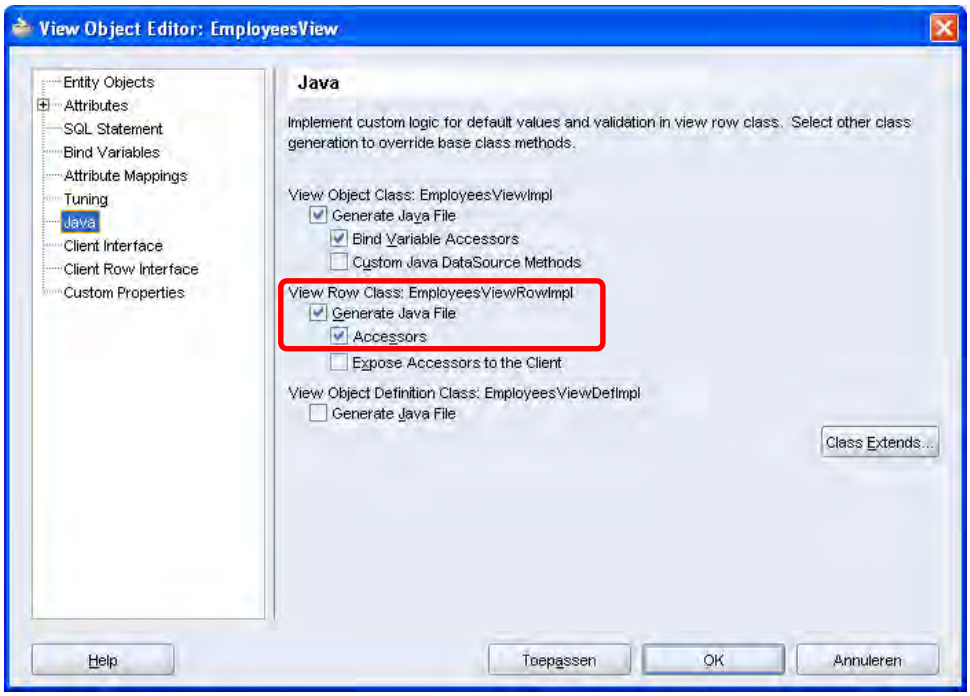
1. Select the View Object, right mouse click, select Edit <ViewObject> to open its Properties dialog.
2. Go to the Attributes node and click on 'New'.
3. Give the attribute an appropriate name.

4. Make sure that the 'Mapped to Column or SQL' checkbox is unchecked.



5. Go to the Java page.

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



7. Press OK

8. Open the generated ViewRowImpl.java file (right-click the View Object and select Go to View Row Class) 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. Save the changes and close the View Object dialog.

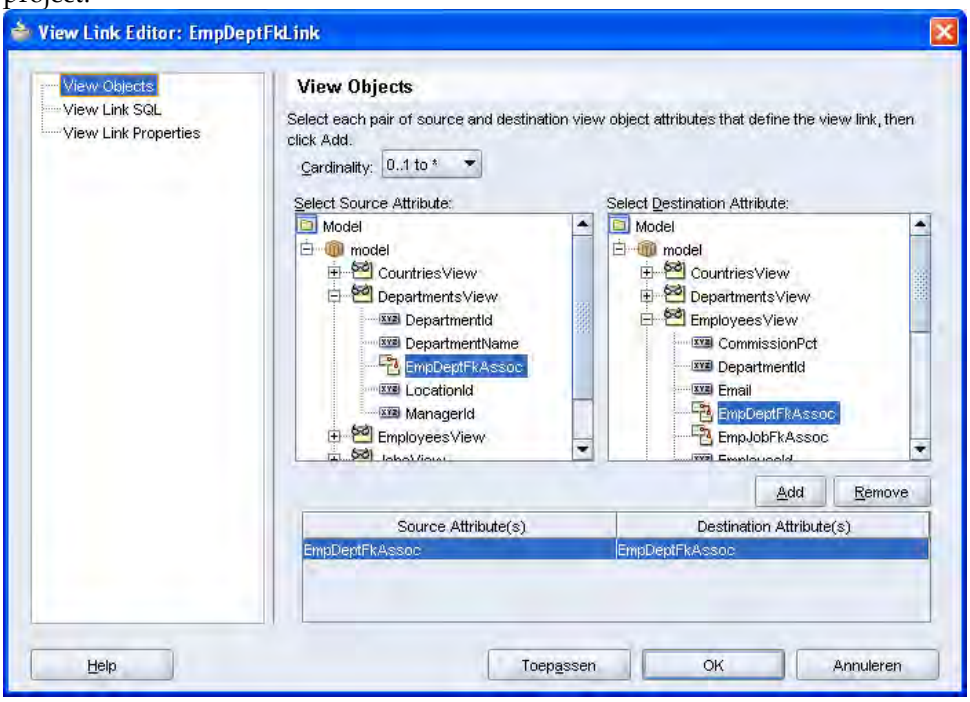
Reference: It is recommended to test the ViewObject with the Business Components Browser. See section Test the model.

3.3.7. Setting Up Master-Detail Synchronization

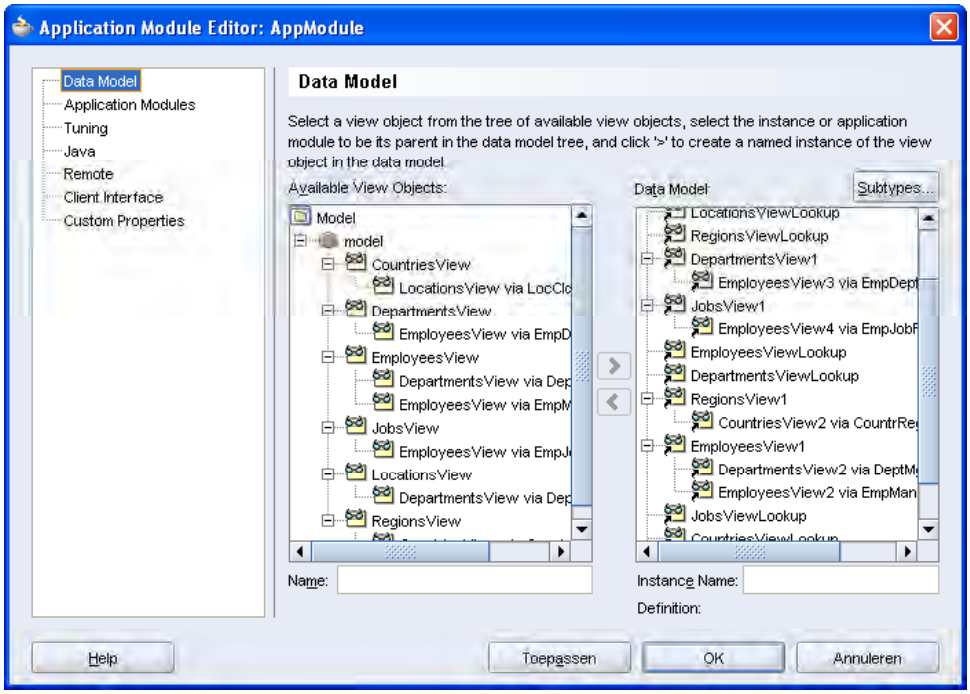
JHeadstart is capable of generating parent-child or master-detail layouts. For example you want to show a department with all the employees in that departments as detail.

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 multiple levels of nesting. For example Regions, consisting of Countries, consisting of Locations and so on. See section 5.6 - Creating Tree Layouts, for an example of deeper nesting.

3.3.8. Defining View Links and View Object Usages for Lookups

JHeadstart is capable of generating dropdown lists or lists of values for entering references to other rows. For example, when entering or updating an Employee, you want to set the Employee's Department by choosing from all available Departments in the database.

When you want to let the JHeadstart New Application Definition Wizard automatically include such lookups, you have to make sure that View Links exist between the relevant View Objects. In the example, a View Link must exist between the Employees View Object and the Departments View Object.

For the purpose of automatically adding lookups, it is **not** necessary to include a usage of the View Link in the data model of the Application Module. The New Application Definition Wizard will automatically add lookup View Object usages in the Application Module.

If you later want to add a new lookup to an existing Application Definition, it is **not** necessary to have any View Links. However, you do need a View Object usage in the Application Module for the lookup data collection. We recommend creating a dedicated View Object usage for lookup purposes, because if the same View Object usage were also used as the main data collection of a page, applying search criteria would result in not having the complete list to choose from in the lookup.

3.3.9. Testing the Model

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

Use the 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 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.

CHAPTER

4 Using JHeadstart

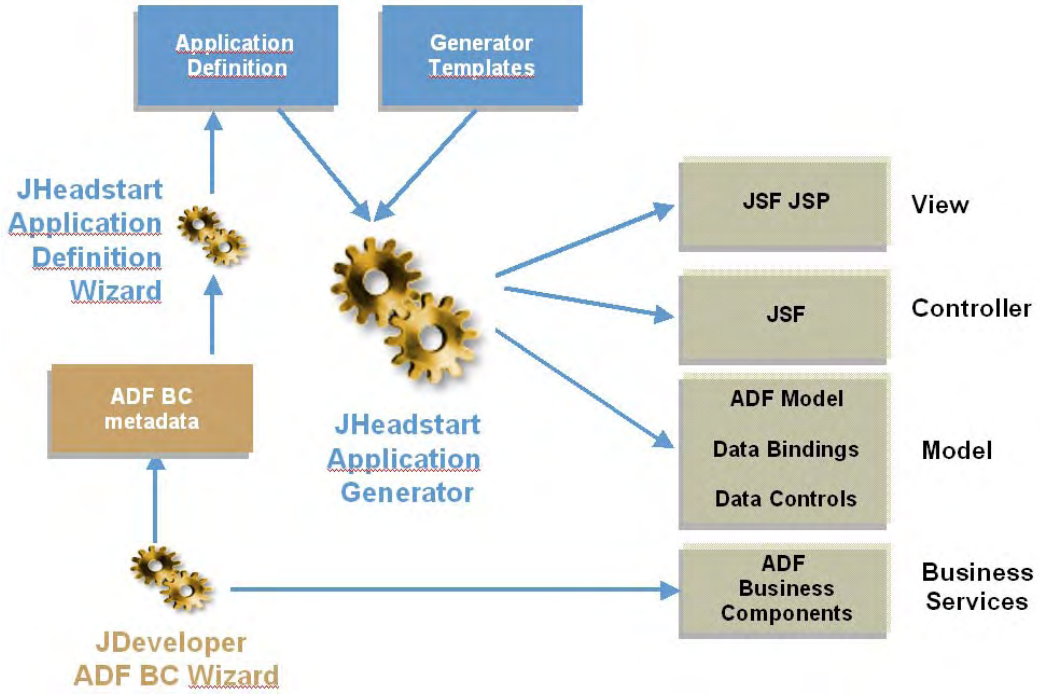
This chapter provides information on how to use JHeadstart in general. The following topics are covered:

- Understanding the Generator Architecture
- Using the JHeadstart Enable Project Wizard
- Using the Create New Application Definition Wizard
- Using the Application Definition Editor
- Running the JHeadstart Application Generator
- Running the Generated Application
- Using Generator Templates
- Generating Mobile Applications
- What Was Generated for What Purpose

4.1. Understanding the JHeadstart Application Generator 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 high-level development process shown in this diagram follows:

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

The Application Definition 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 (ADF Faces with JSP version 2.0 or 1.2).
- The Data Collections that should be displayed and modified.
- The layout styles that should be used to display and manipulate the Data Collections.
- Relationships between the Data Collections: parent-child or lookup.

JHeadstart includes the JHeadstart Application Definition Editor, which is a user-friendly mechanism to edit the Application Definition without having to edit the XML file directly.

4.1.1. Input Output

In addition to the Application Definition, the JAG uses the following inputs:

- JHeadstart Generator Templates

The JAG parses the Application Definition and generates a Model-View-Controller (MVC) application using the following technologies:

- Model: ADF Business Components and ADF Model (data bindings).
- View: JSF JSP and ADF Faces.
- Controller: JSF.

The JAG is capable of generating the following types of output:

- Faces Config files for the JSF Controller.
- JSF JSP files for each displayed page.
- Page Definitons (data bindings) for generated pages.
- Resource bundles for internationalization.
- SQL scripts for populating the JHeadstart database tables when table-driven features are enabled (dynamic menu, flex items, security, internationalization)

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

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

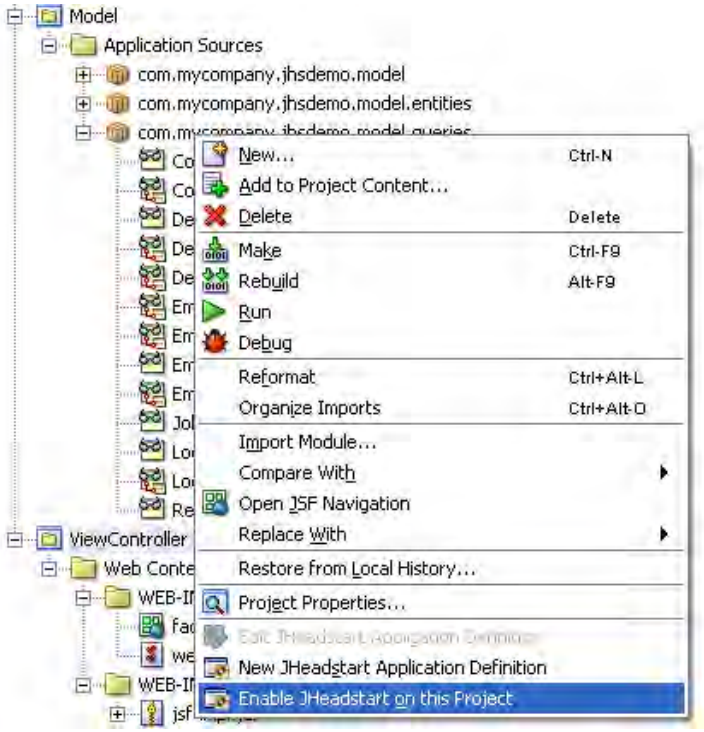
4.2. Using the JHeadstart Enable Project Wizard

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 an ADF Faces page in a project, JDeveloper will automatically add a number of settings to the web.xml file, and add a faces-config.xml file to your project

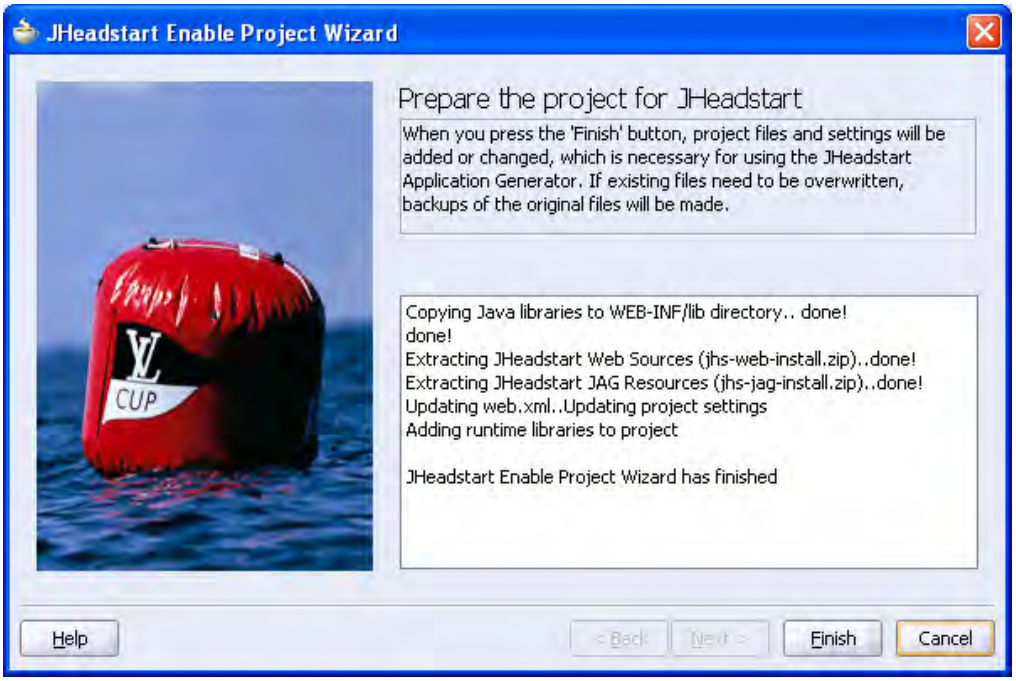
In a similar fashion, the use of JHeadstart also requires such files and settings in your project. We have chosen to make the use of JHeadstart on a project a deliberate choice. Before allowing the use of any JHeadstart Addins on a project, you must first 'enable' JHeadstart on it. Typically, this only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

4.2.1. 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'.



The JHeadstart Enable Project 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:



4.2.2. Enabling JHeadstart on an existing project

The above screenshot is the result of invoking the JHeadstart Enable Project 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:

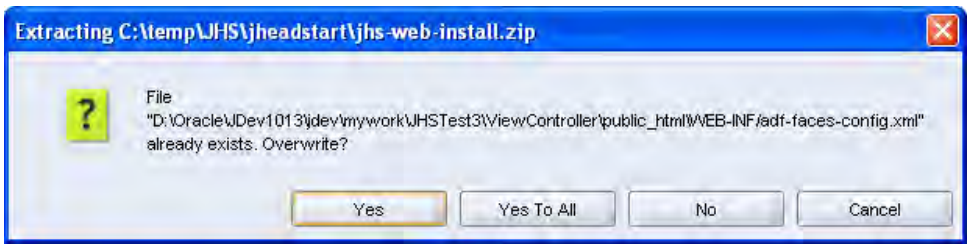
1. Backup the file.

   This is done for files that are absolutely required, for JHeadstart to function correctly, such as 'web.xml' and 'faces-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.

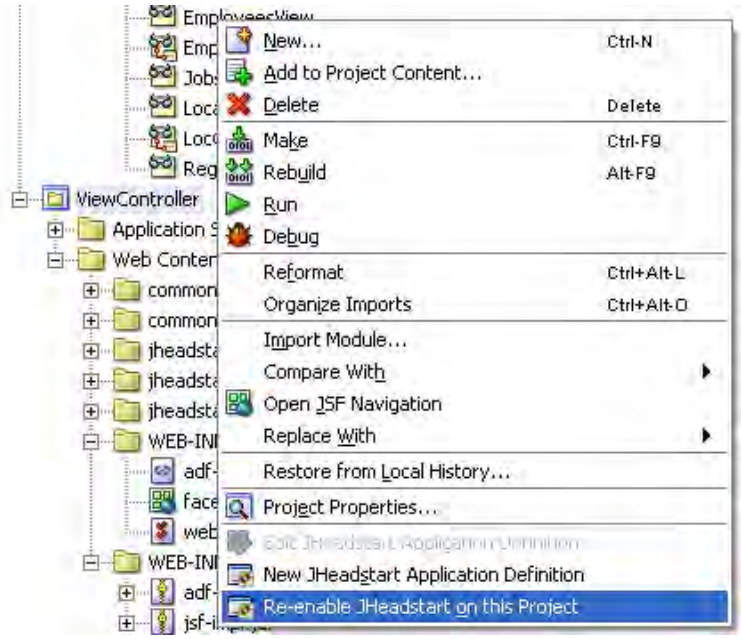
   This is done for all other files, such as Tag Libraries and JHeadstart-specific files. It is unlikely that you made manual changes to these files, so normally you

would choose 'Overwrite All', but you can make your choice to overwrite, backup or ignore on a per-file basis if you want.

4.2.3. 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'.



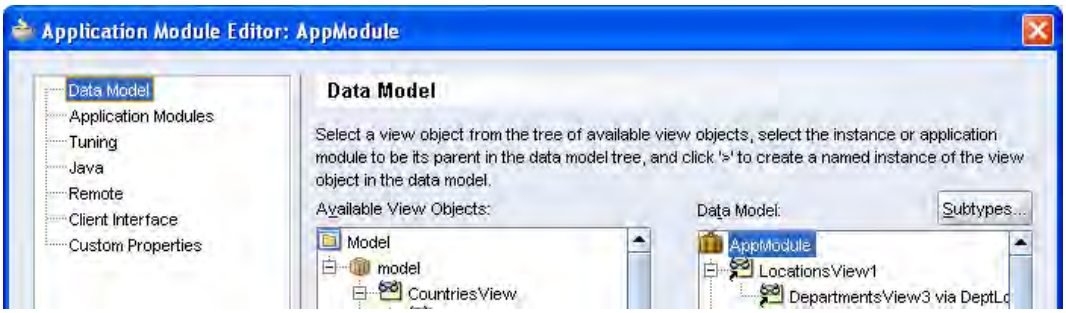
Attention: As you can see, because JHeadstart was already enabled on this project, you can choose the menu option 'New JHeadstart Application Definition', and the menu option to launch the JHeadstart Enable Project Wizard was renamed to 'Re-enable JHeadstart on this Project'.

4.3. Using the Create New Application Definition Wizard

After enabling JHeadstart, you will typically create a JHeadstart Application Definition xml file. You can create a new Application Definition by right-clicking the ViewController project and choosing 'New JHeadstart Application Definition'.

At this point, JHeadstart will make changes to your ADF Business Components Model.

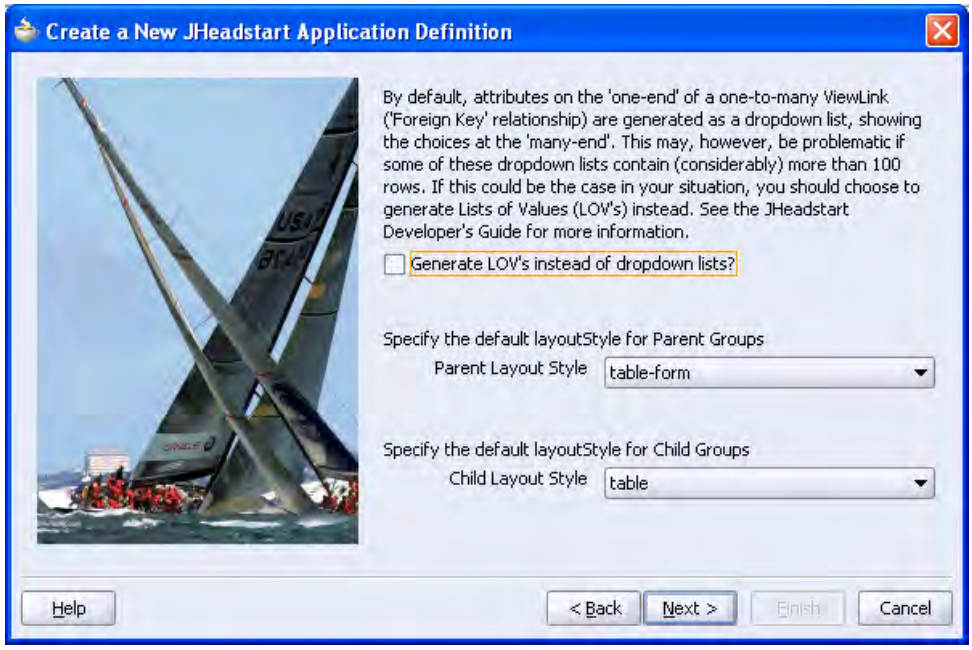
For each View Link, JHeadstart generates a Lookup data collection by default. JHeadstart adds new instances of the ViewObjects to the Application Module with name *Lookup. You can inspect this behavior by editing your Application Module.



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. This way you can perform a search in the Employees maintenance page, without limiting the available values in the lookup.

4.3.1. Dropdown Lists or Lists of Values

One of the questions asked by the New Application Definition Wizard is 'Generate LOV's instead of dropdown lists?'



The problem that could occur if you have a dropdown list with a very large number of rows is twofold. The query performed on the database to retrieve the rows is slow, and the HTML needed for rendering the page becomes very large and takes a long time to load. In extreme situations, this might mean that when trying to show the page, the database hangs and/or the page never shows up.

Rule of thumb: click this checkbox if one or more of the tables you expect to be used for choosing the many-end of a relationship contains considerably more than 100 rows.

Warning: The Application Definition is not visible in your JDeveloper project. Until you pressed the Save All icon in the JDeveloper toolbar.

4.4. Using the Application Definition Editor

The Application Definition 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 (top level) group in the application will be generated as a tabbed page in your web application.

One Application Definition can contain only one Service. If you need multiple Services, you must create multiple Application Definitions.

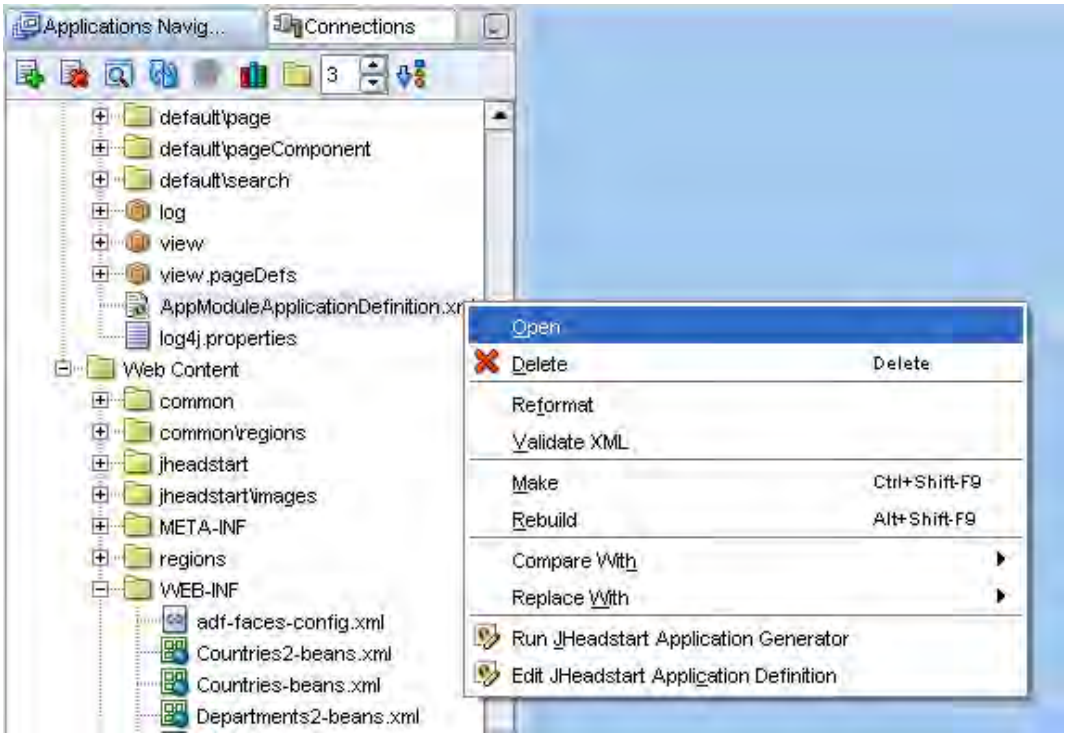
Reference: How to create an Application Definition and how to define a service is described in Chapter 2, *Getting Started*. This section only discusses how to create new groups, and how to modify and remove existing groups.

4.4.1. Maintaining the Application Definition

The JHeadstart Application Definition Editor helps you to maintain the Application Definition 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.

4.4.1.1. Starting the Application Definition editor

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

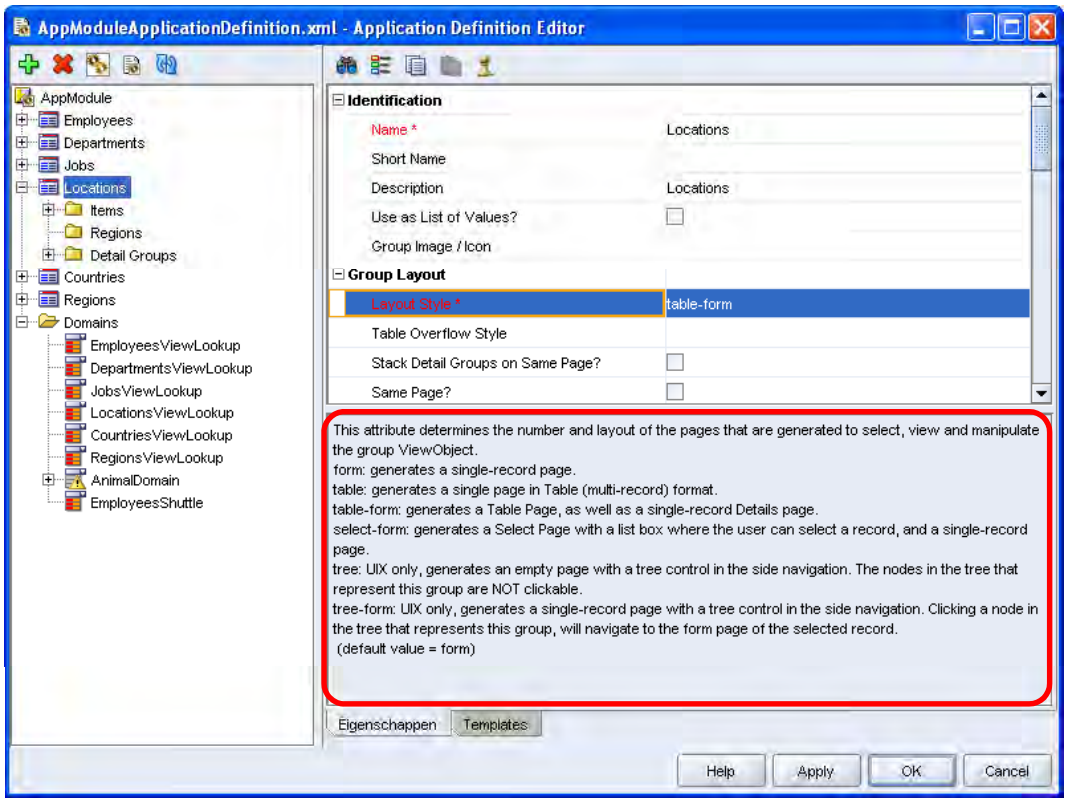


Select 'Edit JHeadstart Application Definition' to open the editor.

4.4.1.2. Using the help in the Application Definition editor

The help in the Application Definition 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 Definition 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:



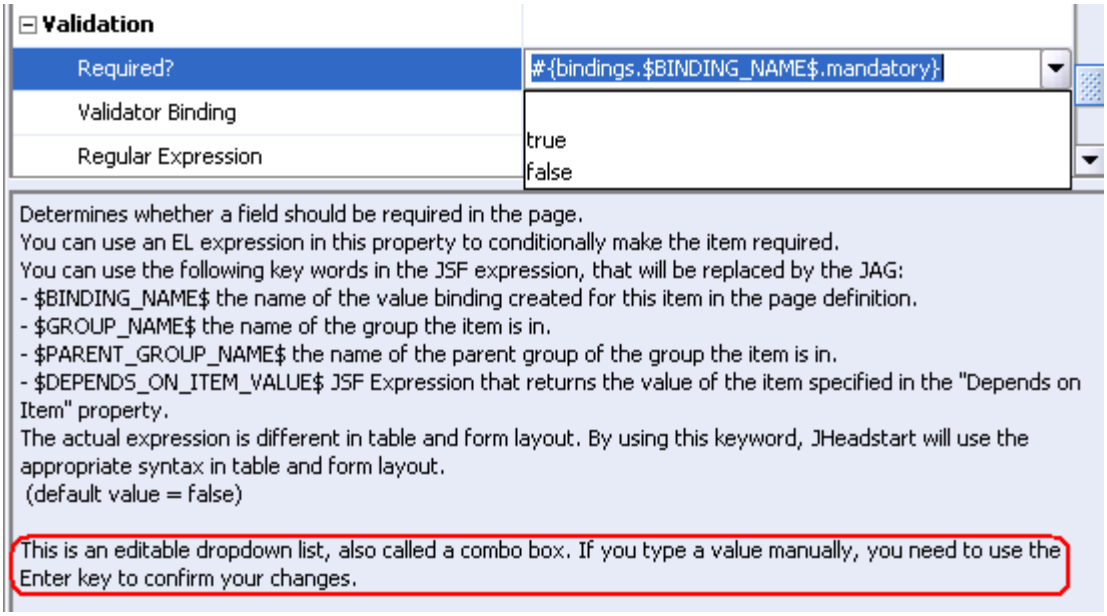
This area may seem unnecessary small. You can increase or decrease the size of this area, as you desire, just by placing the mouse cursor on the line above the help text and move the line up or down.

4.4.1.3. Editing the Properties

There are four types of properties:

1. Text properties
2. Check boxes
3. Dropdown lists
4. Combo boxes or editable dropdown lists

How to edit the first three types is obvious. But the fourth type needs a little explanation.



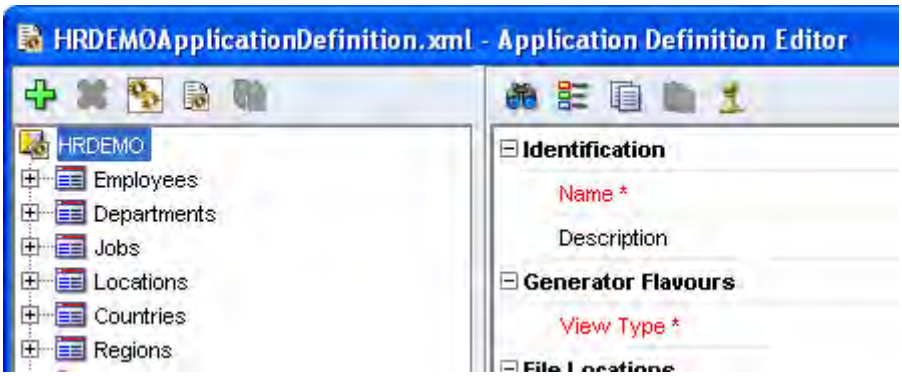
As you can see in the example of the item-level **Required?** property above, a combo box has a dropdown list from which you can choose a value (in this case true or false), but you can also type in a different value. If you do so, you must confirm this typed-in value by pressing the Enter key. This is also mentioned in the online help of the property.

Warning: If you type in a value in a combo box, and then press Tab or click the mouse in a different cell, your typed-in value will be lost. Use Enter to confirm your changes.

4.4.2. 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.

A part of a service definition seen through the Application Definition Editor



Attention: When partitioning the application into services, take into account the following restriction:

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

4.4.3. Groups

A service is made up of one or more groups. A group allows users to query and modify a single data collection that maps to an ADF BC 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.

Compared 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.

A group consists of Items, Regions and Detail Groups. These concepts are discussed below.

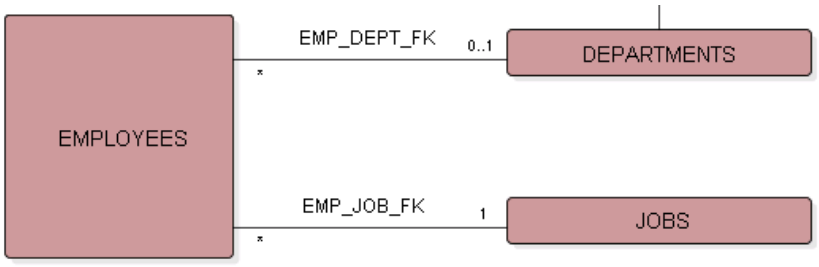
4.4.4. Items

An item is a mapping to an attribute of an ADF View Object (which normally maps to a database column). All kinds of properties can be set for an item. For instance you can specify a default value or a label (used when generating prompts). An item can have a List of Values, which is explained in the next section.

4.4.5. Lists of Values

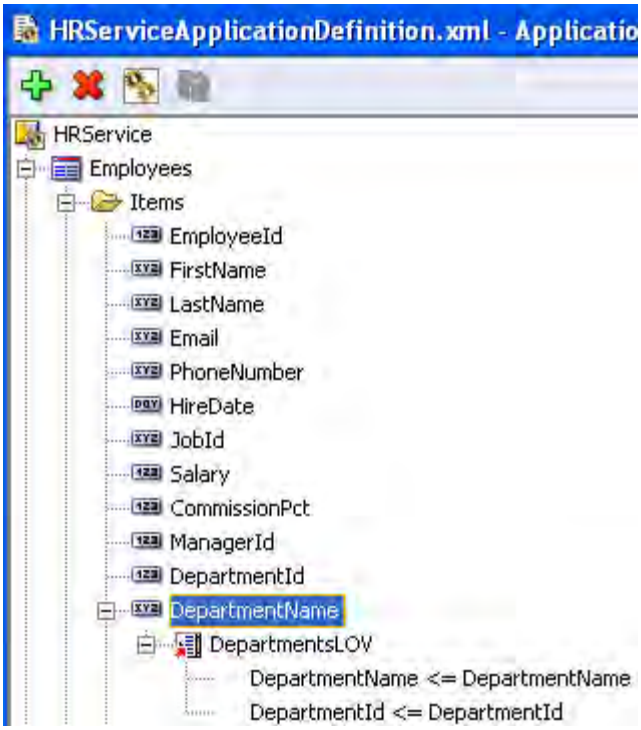
A List of Values (LOV) construction in an Application Definition links an item to a group of type LOV, and specifies which items in the LOV group are to be mapped to which items in the base group. This is required if in the generated application you want to populate the item using a List of Values popup window instead of using a dropdown list.

Example The EMPLOYEES table has a foreign key to the DEPARTMENTS table.



When adding an EMPLOYEE, you need to choose the department. Suppose you want to enter the the department using a List of Values, and you want to put that LOV on the Department Name instead of the Department Id.

In the Application Definition you then have to create an item DepartmentName (after first creating it in the ADF BC View Object), and link an LOV to that item. This is how it looks in the Application Definition editor:



An LOV is populated by means of an LOV group. This is a (top level or nested) group with the property **Use as List of Values?** checked.

Within the LOV one can create a mapping of source and target items. In other words which item of the LOV group should map to which item of the current group. The target of the first value always automatically maps to the currently selected item.

4.4.6. Regions

The regions folder can contain three different types of objects. If you add a new Region you can choose which type you want to create.

1. Item Regions
2. Detail Group Regions
3. Region Containers

An **Item Region** allows you to group items into a named section (region) on a page. You can define as many item regions as you want for a group.

A **Detail Group Region** can be used to create a dedicated region for a detail group (nested group). This way one has more control over the placement of the detail group on the page.

Attention: The detail group(s) inside the Detail Group Region must have the property **Same page?** checked. Only such groups appear in the dropdown list of the Group Name property (of the Detail Group Region).

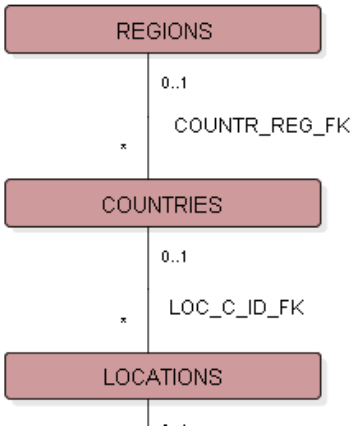
A **Region Container** is, as the name says, a container for regions. The Regions folder is an example of a Region Container. This is where the **Layout Style** can be set. Three different flavors can be used: horizontal, vertical and stacked.

Some screenshots can be found in the section Using regions.

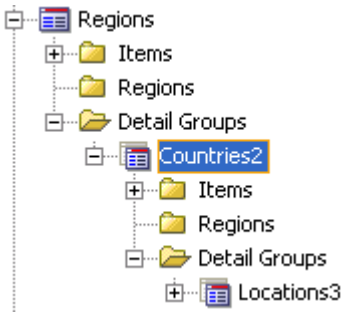
4.4.7. Detail Groups

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

Example `A Region can have one or more Countries, and a Country can have one or more Locations.`



`This structure can be reflected in the Application Definition like this:`



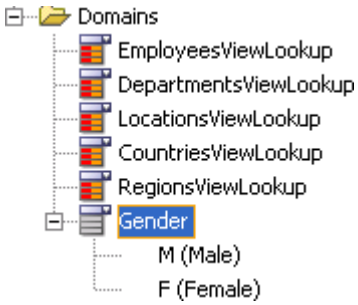
Depending on the layout options you choose, you can display a master group and its detail(s) on different pages or on a single page.

Attention: JHeadstart has no restrictions on the maximum level of nesting of groups.

4.4.8. Domains

A domain is a (short) list of values normally used to populate a dropdown list. Two kinds of domains are distinguished: static and dynamic.

A static domain is nothing else than a list of hard coded domain values. See the Gender domain in the screenshot below.

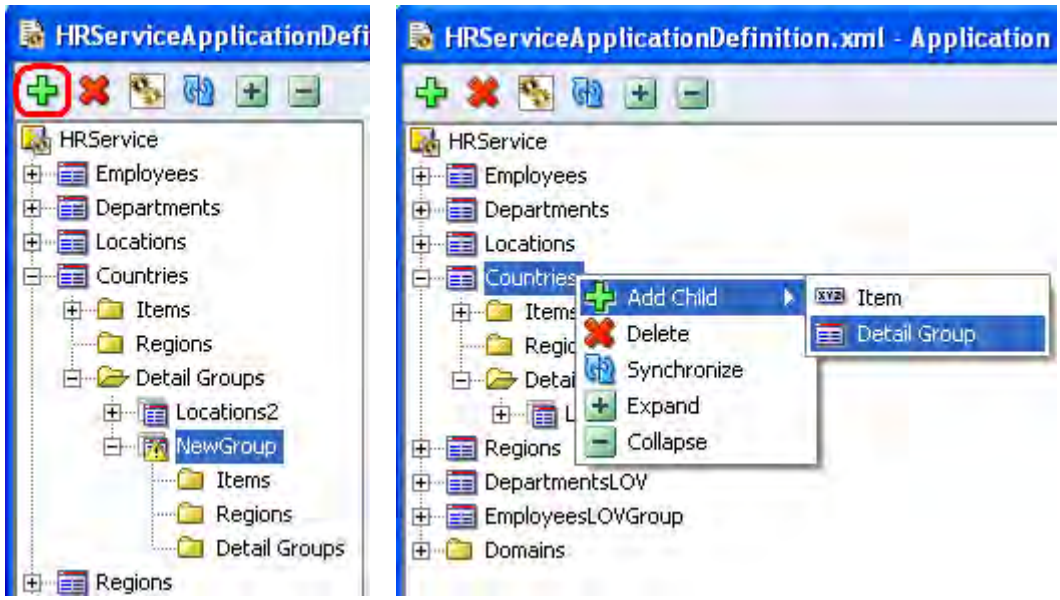


A dynamic domain is a domain based on an ADF View Object and therefore based on a query. See for example the EmployeesViewLookup domain in the screen shot.

Items with **Display Type** dropDownList, radio-horizontal and radio-vertical have a **Domain** property where you can specify a static or dynamic domain.

4.4.9. Manipulating Objects

You can create the objects described above by using the green plus (+) symbol in the upper left corner of the Application Definition Editor, or by using the right-mouse-click menu in the left hand panel. Just select the intended parent node in the tree on the left and press the plus symbol, or right-click and choose Add Child. Whenever it is unclear what type of object should be created, a list is shown and the user can select the desired type. Otherwise the only possible type of object is created.



The table below shows what object types can be created when a certain type of node is selected. The last column indicates the name of the newly created object.

Node	Object Type	Name
Service	Base Group	NewGroup
Base Group / Detail Groups folder	Item	NewItem
	Detail Group	NewGroup

Items folder / Item Region	Item	NewItem
Item	LOV (+ one LOV Value)	Choose a LOV Group (+ [currentItem] <= undefined)
LOV	LOV Value	undefined <= undefined
Regions folder / Region Container	Region Container	NewRegionContainer
	Detail Group Region	NewGroupRegion
	Item Region	NewItemRegion
Domains folder	Static Domain (+ one Domain Value)	NewStaticDomain (+ undefined)
	Dynamic Domain	NewDynamicDomain
Static Domain	Domain Value	Undefined

After creating a new object, its properties have to be set. A red property is mandatory (also indicated by a * at the end of the label) and a black one is optional.

4.4.9.1. Moving objects

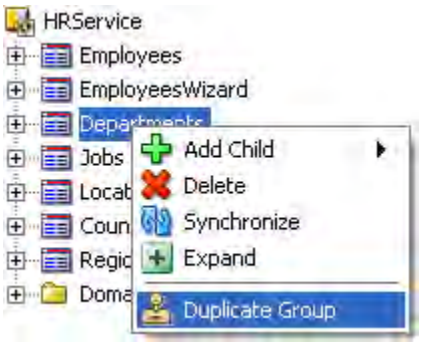
You can move an object in the Application Definition to a different position under the same parent by dragging and dropping the object. This way you can influence the order in which for example level 1 menu tabs or fields in a form layout are generated.

You can also move objects to a different parent, if that parent is capable of holding objects of that type. For example, you can move a detail group to another top-level group, or you can move an item from a group to an item region.

4.4.9.2. Copying objects

There are two ways for duplicating objects in the Application Definition.

1. You can right-click an object in the Application Definition Navigator and then choose Duplicate <object type>.



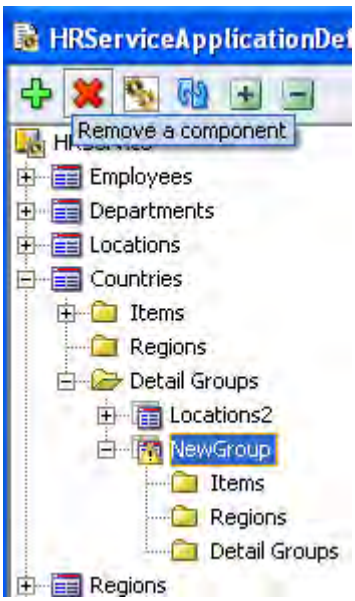
The new object is an exact copy of the original, except for the name, which is "Copy of <original name>".

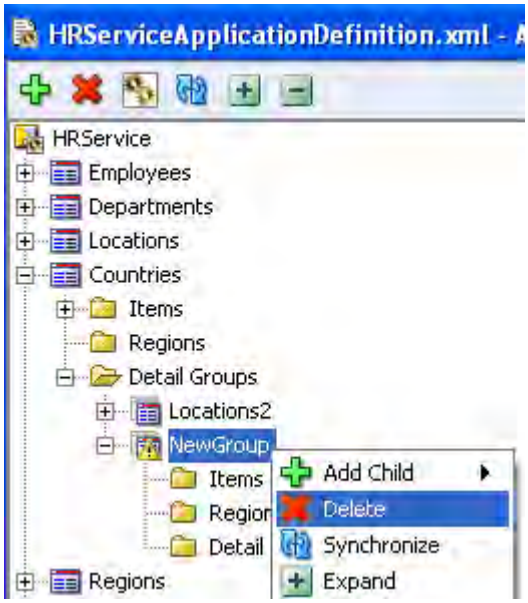
2. You can also copy an object in the Application Definition Navigator by dragging the object while holding the Ctrl key. The cursor will be decorated with a plus (+) in a box to indicate that the dragged object will be copied instead of moved.

   When releasing the Ctrl key a duplicate of the original object will be created. The new object is an exact copy of the original (including the name property). The new object will be created as child of the object currently under the cursor.

4.4.9.3. Deleting objects

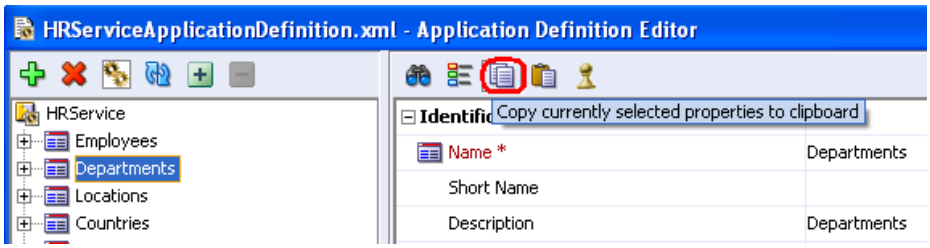
You can also quickly delete objects using the editor. Simply select the object you want to delete, and press the red cross (x) icon in the tool bar or in the right-mouse-click menu:



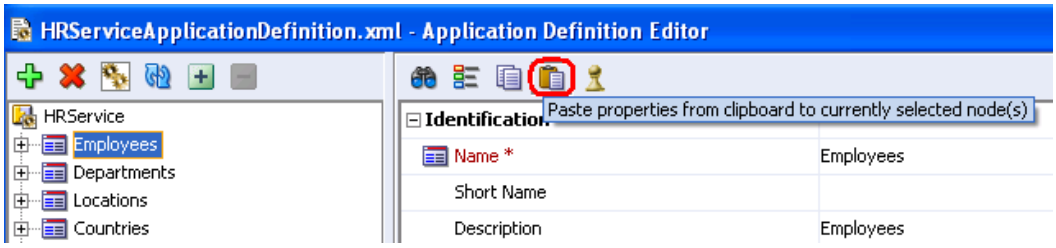


4.4.9.4. Using the clipboard to copy and paste multiple properties

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):



4.4.10. Novice Mode and Expert Mode

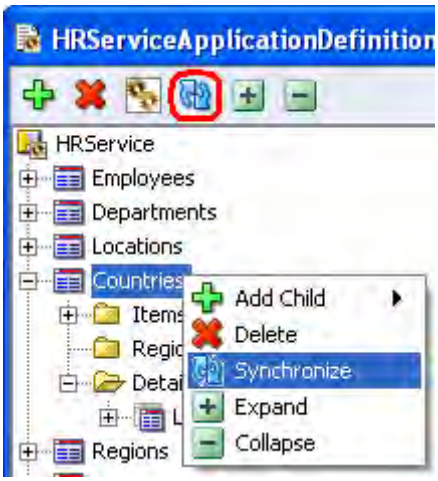
The Application Definition Editor gives you the possibility to change between novice and expert mode.

In novice mode only the most relevant properties are displayed. Which properties are relevant, might depend on the value of other properties. For example, if you change the Layout Style from 'form' to 'table', the Form Layout properties are hidden and the Table Layout properties become visible.

You can switch to expert mode by clicking the icon on the upper right of the editor, as highlighted in the screen shot above. In expert mode you can see all properties, regardless if they are applicable or not.

4.4.11. Synchronize View Objects with groups

In best-case scenarios View Objects never change, but in real life they do. Therefore the synchronize button is added to make life a little easier. Whenever the attributes in a View Object change, one can select the corresponding group in the Application Definition editor and press the synchronize button as highlighted below, or by using the right-mouse-click menu and choosing Synchronize. This action will add/remove all missing/redundant items in the group.


HRServiceApplicationDefinition
HRService
Employees
Departments
Locations
Countries
Items
Regic
Detai
Regions
Add Child
Delete
Synchronize
Expand
Collapse

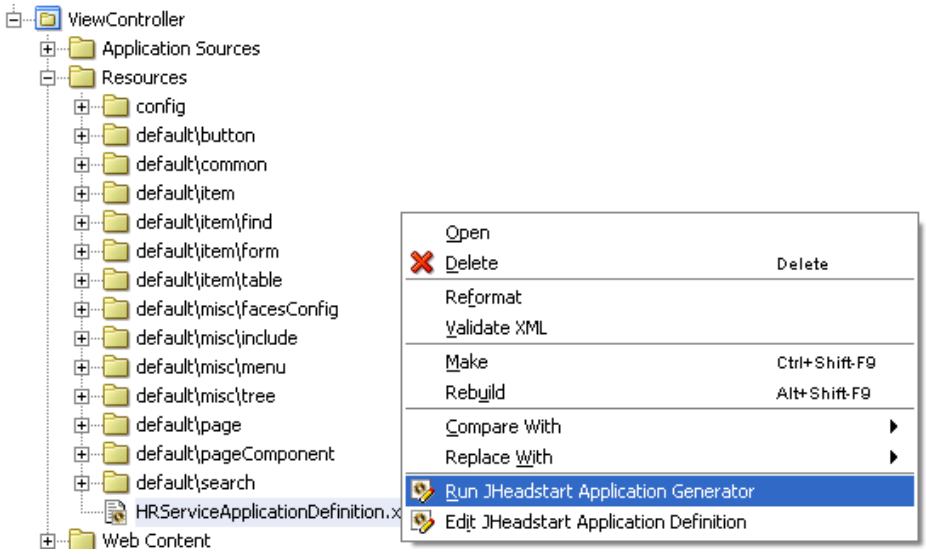
4.5. Running the JHeadstart Application Generator

Before you start generating your application, make sure you have applied the naming conventions and other service-level settings as discussed in chapter "Team-based Development", section "Organizing JHeadstart Application Definition Files".

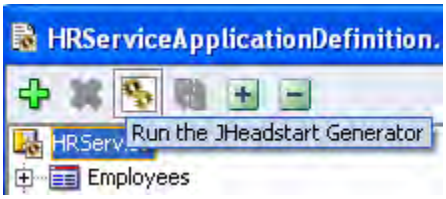
Once you have got the service and group definitions right, you can generate the application.

There are two ways to start the JHeadstart Application Generator (JAG):

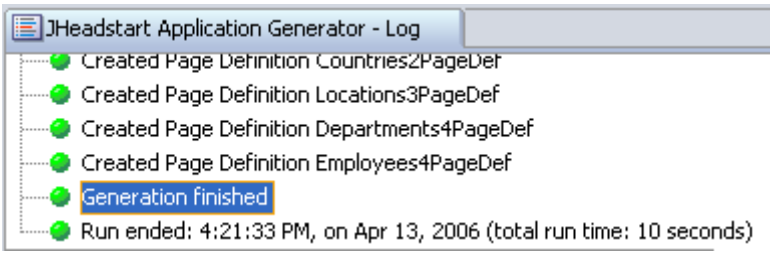
1. Right click the Application Definition File in the Applications Navigator, and choose Run JHeadstart Application Generator.



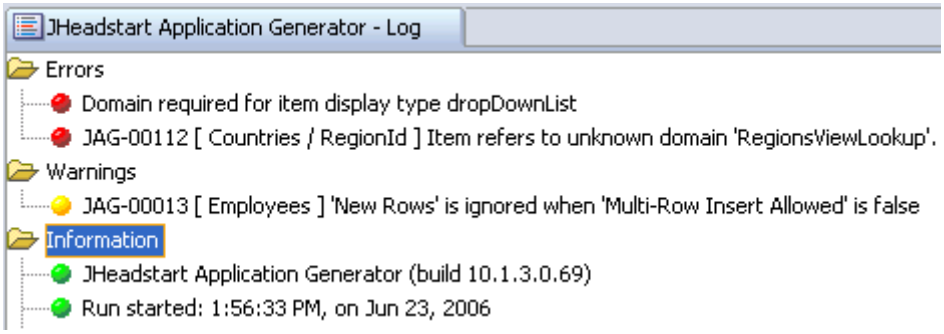
2. From within the JHeadstart Application Definition editor, click the third button from the left.



Now JAG will generate the View and Controller layers of the application, including the ADF Model data bindings to the ADF Business Components. The progress is logged in the Jheadstart Application Generator – Log.



You will see logging of what has been generated (the **Information** messages).
If a (potential) problem is detected that does not prevent the Generator from doing its job, you will see **Warnings**.
Finally, if the application cannot be generated, you will see **Errors**.

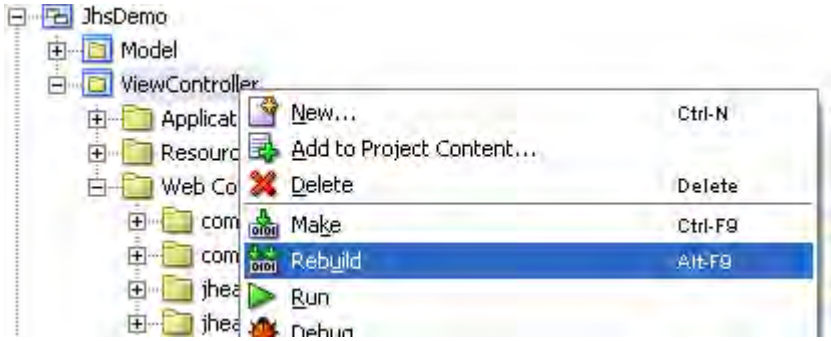


4.6. Running the Generated Application

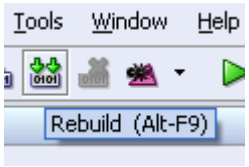
When the JHeadstart Application Generator has completed successfully, you can run and test your application.

Before running the generated application, it is a good habit to always rebuild it first (causing the project files to be copied to the class path). You can rebuild it in one of the following ways:

- Right click the ViewController project in the Navigator, and choose Rebuild



- Click the Rebuild icon in the JDeveloper toolbar



You can run the generated Application in one of the following ways:

- Run the ViewController project (using right-mouse-click in the Navigator or using the Run button in the JDeveloper tool bar),

  The JHeadstart Application Generator has automatically set the first generated page of the first group to be the default run target

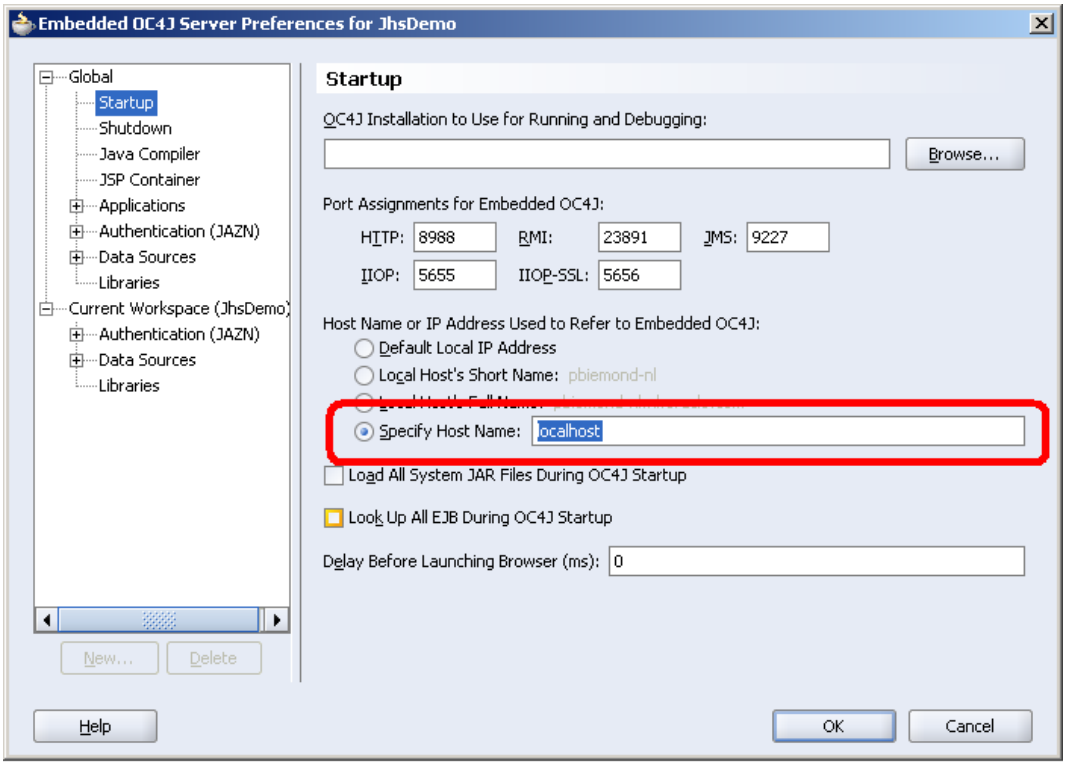
- To directly run a specific page, select one of the generated .jspx files in the Application Navigator, right-mouse-click and choose Run.

- Open the 'faces-config.xml' in Diagram view and right click a .jspx page to run.

4.6.1. TroubleShooting

If the application 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.



4.6.2. Dealing with Code Segment Too Large Error

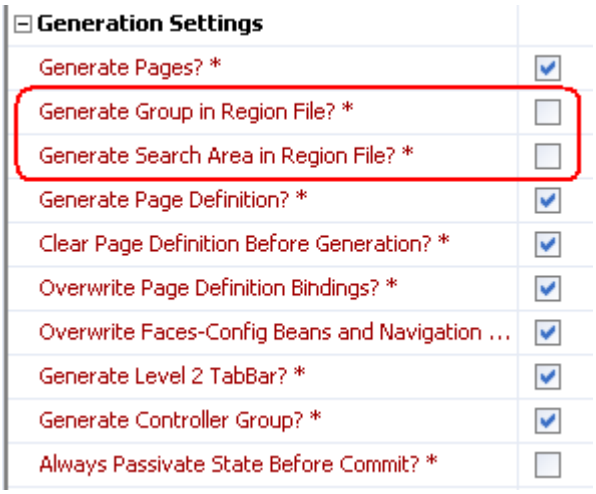
If you have generated large pages, with many groups on the same page, and/or groups with many items, compilation of such a page might fail with an error like this:

Error: code segment of method _jspService(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse) too large

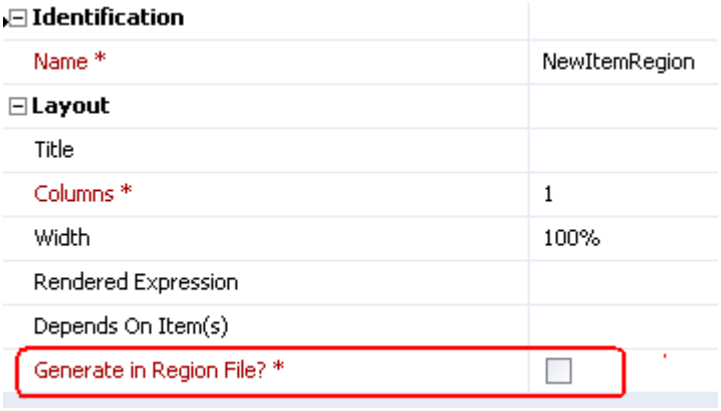
This error is caused by a limitation in the Java language. The content of a Java method cannot be larger than 64KB. With a very large JSF page, the compiled servlet might get a method (like _jspService) that is too big to be compiled.

JHeadstart provides an easy work around for this error: you can move part of the content of the page to separate ADF Faces region files that are included in the orginal page at runtime, very similar to the concept of a JSP Include. The look and feel and behavior of the page remains unchanged, only the way the page is composed at runtime is different. You can use three properties in the Application Definition Editor to determine which part of the page is generated into a separate ADF Faces Region:

- Group property **Generate Group in Region File?**
- Group property **Generate Search Area in Region File?**
- Item Group Region property **Generate in Region File?**



Note that all three properties are only visible in expert mode. Typically, when you run into this error, you first start generating whole groups on the page in a separate ADF faces region by checking the **Generate Group in Region File?** checkbox for one or more groups on the page. In most situations, this will solve the problem. However, you might have one very big group on the page with very many items. If most of these items also appear in the advanced search area of the group, you can check the checkbox **Generate Search Area in Region File?**. If the problem still persists, the only solution is to divide the items over multiple item regions, and at the item region container, check the checkbox **Generate in Region File?**.



4.7. Customizing Using Generator Templates

This paragraph discusses how you can use custom generator templates to implement functionality in your application that cannot be generated using the default templates.

The following topics are discussed:

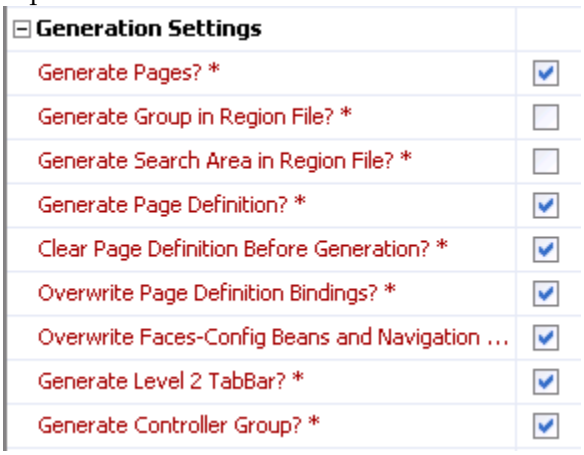
- Recommended approach to customizations
- Introduction into the JHeadstart template structure and how to configure JHeadstart to use your custom templates
- Introduction into the Velocity Template Language used by JHeadstart
- How to apply common customizations

4.7.1. Recommended Approach for Customizing JHeadstart Generator Output

It is important to understand that the artifacts produced by JHeadstart are fully ADF compliant, and implement numerous ADF best practices available on the internet. When you use ADF drag and drop, ADF creates code snippets in JSF pages, page definitions and bindings within these page definitions, and managed bean definitions. All these artifacts are also created by JHeadstart. At any time in your development process you can start using the visual design-time tools and code editors in JDeveloper to implement functionality that cannot be generated out-of-the-box.

Now, if you start customizing a generated page, page definition or faces-config file, and then generate your application again, you would loose the changes again. So, you have three choices once you start customizing JHeadstart-generated output:

1. Do not use the JHeadstart Application Generator anymore on the application definition that produced the output you customized.
2. Switch off generation of the files you modified. Both at the service-level and at the group level you have generator switches that you can use to turn off specific output. The screen shot below shows these group-level switches in the Application Definition Editor. Note that these properties are only visible in expert mode.



3. Move the customizations to custom templates, configure JHeadstart to use your custom template, and keep on generating.

You are free to choose whatever option suits you best, but we would like to share our own opinion and the experiences of many JHeadstart customers before you make a decision:

- The first option implies that you only use JHeadstart in the beginning of your project to get a "head start". While this is in line with the name of the product ☺, this is the least attractive option in our view. When requirements change for any page in the application definition, even pages that are not customized, they need to be implemented manually. In short, developer productivity will decrease quickly and dramatically with this approach.
- The second option is easy and fast. It is a good option if you do not expect (significant) changes in the customized output. The question here is, how reliable are your expectations? In modern agile application development methods changing requirements are the rule rather than the exception. If changes are needed, for example as a result of a database change, then applying these changes manually will decrease developer productivity as well. Note that even apparent simple UI changes like changing a drop down list into a list of values easily take hours if not days to implement manually.
- The third option is initially a bit more work and requires you to understand the JHeadstart templating architecture (explained in the remainder of this paragraph). We have seen quite a few customers that initially decided to go for the second option. However, once they discovered how easy and fast it is to perform this additional step of moving custom code to a custom template, they consistently chose for the third option. This fact is best illustrated by a survey we conducted amongst JHeadstart customers that showed that the vast majority was able to keep their application 100% generatable. This might sound unrealistic, but when you realize that *all* content of the generated pages and faces-config files is 100% driven by generator templates, meaning you can really customize anything you like, you will better understand the outcome of this survey. Note that a powerful side-effect of this approach is that you automatically "document" your customizations by creating a separate tree of custom templates. This is easy for maintenance, another developer can quickly identify and understand the customizations, and allows for a smooth transition when migrating to a new JDeveloper/JHeadstart version. For example, when JDeveloper/JHeadstart release 11 is available, then regenerating your existing application with JHeadstart 11 will automatically leverage the ADF Faces Rich Components. You only need to modify your custom templates to leverage the new release 11 features. When choosing option 1 or 2, the page customizations are "hidden" in the customized page, and it will take a lot more effort to identify and upgrade these customizations.

4.7.2. Using Custom Templates

When generating, JAG replaces placeholders in the templates with content taken or derived from the Application Definition.

The JAG uses Velocity, an open source Java-based template engine from the Apache Foundation. See also the paragraph Velocity and the Velocity Template Language below.

The JHeadstart templates are stored in the templates directory of your JHeadstart project. This folder contains the following files:

- **config/jag-config.xml**: configurable settings for the JHeadstart Application Generator

- **config/defaultTemplateBindings.jtp**: JHeadstart Template Properties file that defines which Velocity template files are used for what purpose

- **default/*/*.vm**: default Velocity template files used for generating the application

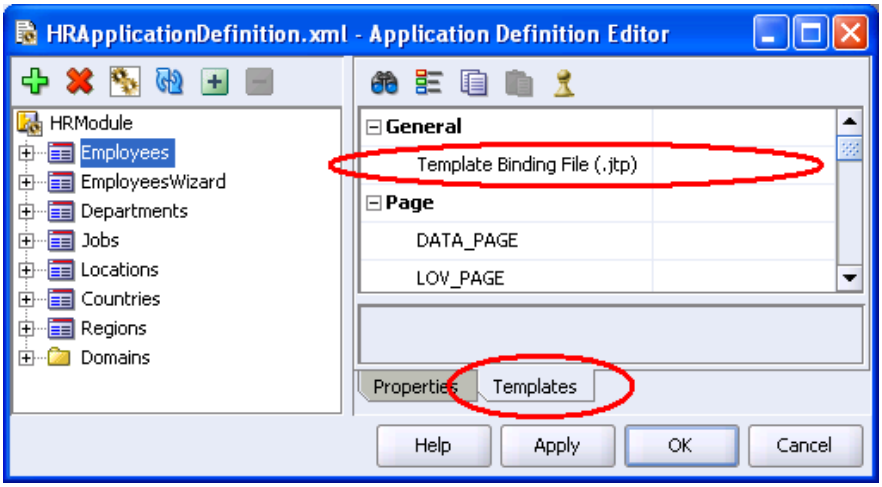
Warning: Never customize one of the standard JHeadstart files directly, be it a default template, a.jspx, or any other file that was created by enabling JHeadstart for your project. When you upgrade to a newer version of JHeadstart they will be overwritten. Always create a custom template template and refer to custom versions of the standard files there. Then you can refer to your custom template in the Application Definition (see below).

Suggestion: Always put your custom templates into a different root folder, so you quickly can have an overview of all template customizations in your project.

The defaultTemplateBindings.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, create a new Template Binding File (for example customTemplateBindings.jtp) and refer to that in your Application Definition.

You can define a custom Template Binding File to be used for the whole service, or for a group, or for any other level. You specify it by going to the Templates tab (instead of the Properties tab) and setting the **Template Binding File** property to customTemplateBindings.jtp.

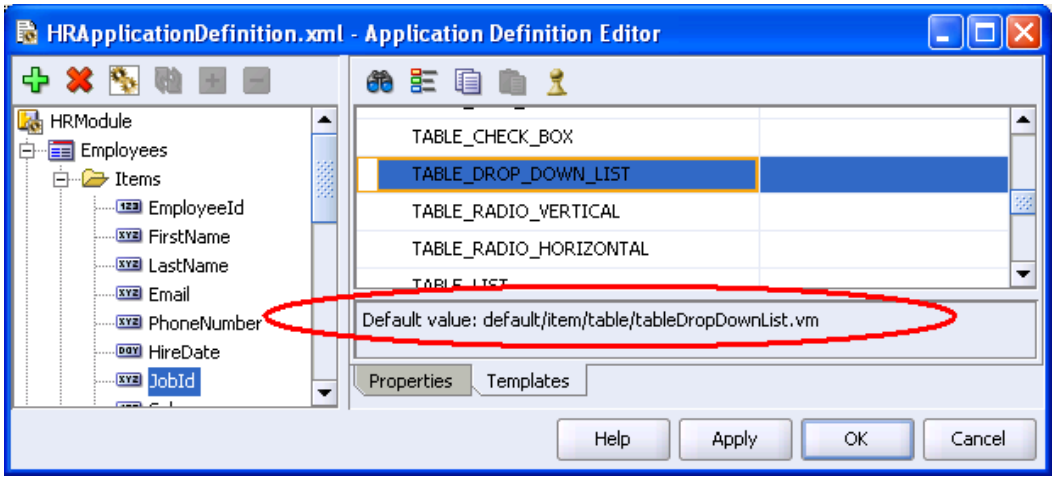
Attention: The Template Binding File you specify in your Application Definition does not need to include the complete list of templates. Only include the lines for the templates you have customized. The other templates will be inherited from the higher level Template Bindings file, or if there is no higher level, from the JHeadstart default settings.



As you can see in the screen shot, you can also override the individual templates that are referred in the Template Bindings File, like DATA_PAGE or LOV_PAGE.

On individual groups, region containers, regions and items, you can override the service-level template settings. You can choose to either specify a custom Template Bindings File for a group or item, or to override an individual template for a group or item.

Suggestion: To easily find out what the default name of a certain template is, check the online help of the template override property.



4.7.3. Finding Out Which Generator Templates Are Used

The service-level checkbox property **Show Template Names In Source** is handy to find out which Generator Templates JHeadstart uses for the various parts of your generated pages. When checked (the default) you will see that the templates used are included as comments in the generated file, when you click the Source tab:

```
<!-- DEBUG:BEGIN:TABLE_TEXT_INPUT : default/item/table/tableTextInput.vm -->
<af:column sortable="true" noWrap="true"
           sortProperty="FirstName">
  <f:facet name="header">
    <af:outputLabel value="FirstName"
                    styleClass="af_column_header-text"/>
  </f:facet>
  <af:inputText id="EmployeesFirstName" value="#{row.FirstName}"
                required="#{bindings.EmployeesFirstName.mandatory}"
                rows="#{bindings.EmployeesFirstName.displayHeight}"
                columns="#{bindings.EmployeesFirstName.displayWidth}"
                maximumLength="20" readOnly="true"></af:inputText>
</af:column>
<!-- DEBUG:END:TABLE_TEXT_INPUT : default/item/table/tableTextInput.vm-->
```

In the example screen shot above you can see that for the FirstName column in EmployeesTable.jspx, the TABLE_TEXT_INPUT template is used which maps to the default template default/item/table/tableTextInput.vm.

4.7.4. Velocity and the Velocity Template Language

The Velocity Template Language (VTL) is meant to provide the easiest, simplest, and cleanest way to incorporate dynamic content. Even a developer with little or no programming experience should soon be capable of using VTL.

VTL uses *references* to embed dynamic content, and a variable is one type of reference. Variables can be set using a VTL *statement*. Here is an example of a VTL statement:

```
#set( $a = "Velocity" )
```

The following rule of thumb may be useful to better understand how Velocity works: **References begin with $ and are used to get something. Directives begin with # and are used to do something.**

The #*macro* script element allows template designers to define a repeated segment of a VTL template, called a Velocimacro. JHeadstart has defined several macros like #ADVANCED_SEARCH_ITEMS. You can find the macro definitions in the project subfolder templates/default/common.

A single line comment begins with ## and finishes at the end of the line. Multi-line comments begin with #* and end with *#.

Text that is not interpreted by the Velocity Template Engine is copied literally.

Reference: See the following three documents at the Velocity website:

User's Guide	http://jakarta.apache.org/velocity/docs/user-guide.html
Developer's Guide	http://jakarta.apache.org/velocity/docs/developer-guide.html
Reference Guide	http://jakarta.apache.org/velocity/docs/vtl-reference-guide.html

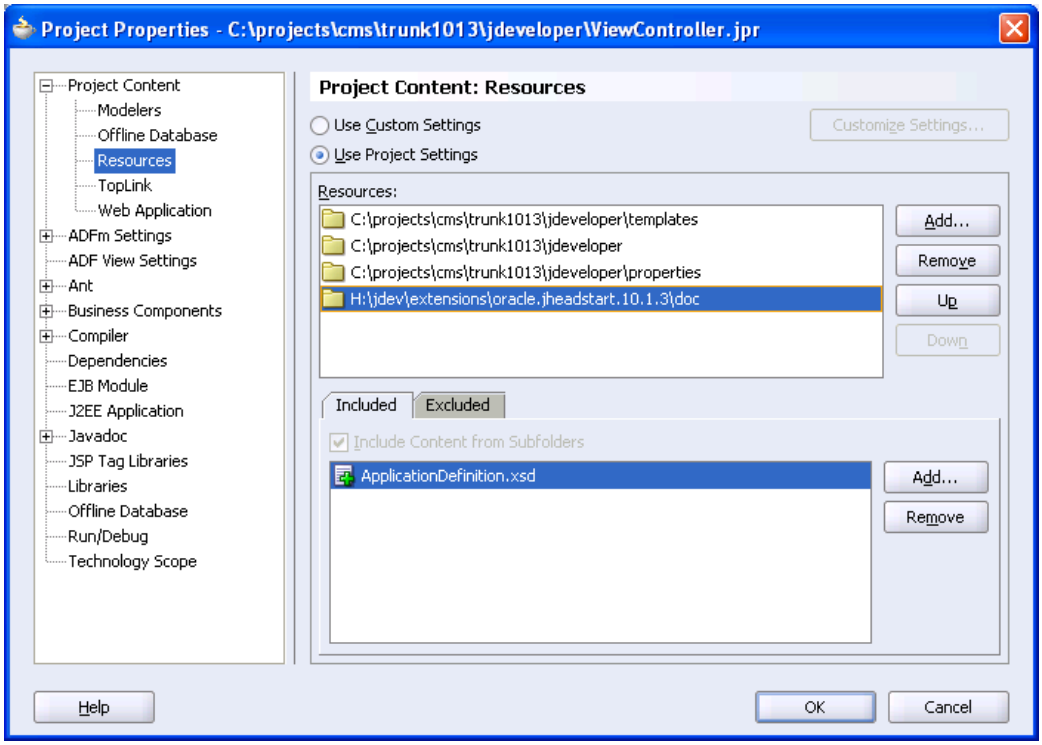
4.7.5. JHeadstart specific constructs in the Velocity Templates

In addition to the normal Velocity capabilities, JHeadstart has created some Velocimacros and other constructs that you can use in your custom templates.

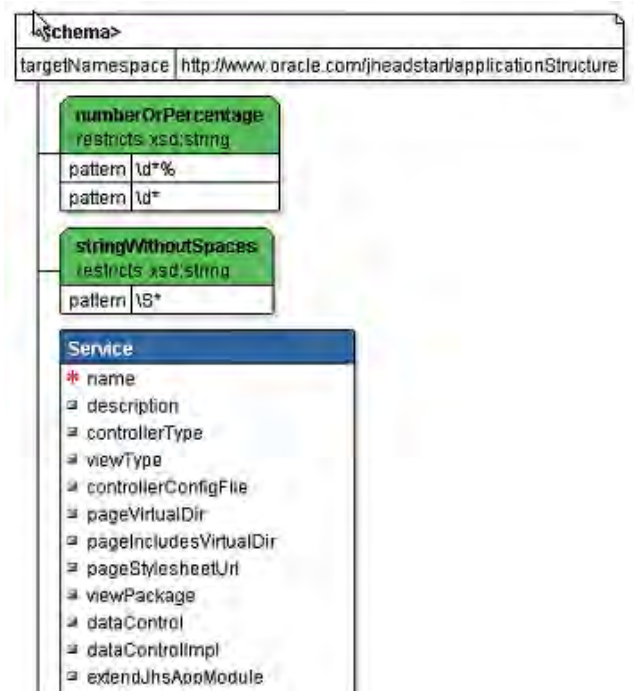
You can access all metadata elements you enter through the Application Definition Editor:

- ${JHS.service}
- ${JHS.current.group}
- ${JHS.current.item}
- ${JHS.current.regionContainer}
- ${JHS.current.itemRegion}
- ${JHS.current.groupRegion}
- ...

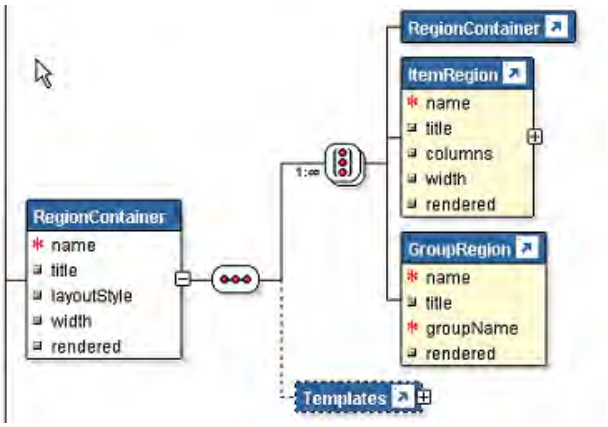
You can use all the attributes of these elements, for example **${JHS.current.group.name}**. For the proper attribute names, see the XML Schema of the Application Definition at [JDEV_HOME]\ jdev\extensions\oracle.jheadstart.10.1.3\doc\ ApplicationDefinition.xsd. You can add this file to your ViewController project like this:



If you then open it you see something like this:



And you can expand the structure to get something like this:



In addition to the XSD attributes, you can access aditional "convenience" methods in so-called PG Model Classes (PG = Page Generator). Each Element Type in the Application Definition has a corresponding PG Model class:

- PGGroupModel
- PGItemModel, etc.

Example: PGGroupModel has convenience method getParentGroup, so you can use `${JHS.current.group.parentGroup.name}`. Since a group can lead to multiple pages, or parts of a page, two additional model classes are created when running the JAG: PGPageModel

- PageModel
- PageComponentModel

For each generated page, an instance of PageModel is created, to which you can refer to using the VTL reference `${JHS.page}`. A PageComponentModel instance is created for each group displayed on the same page. So, if you have a parent group with layout style "form", and two detail groups with Same Page checkbox checked, three PageComponentModel instances are created. You can refer to the current PageComponent using the VTL reference `${JHS.current.pageComponent}`.

Reference: See the Javadoc for all available classes and properties: in JDeveloper choose Help | JHeadstart Documentation Index, and then click the hyperlink 'Javadoc of the JHeadstart Application Generator'. Look for classes like PGGroupModel.

Suggestion: For debugging purposes, you can temporarily add macro `#MODEL_POINTER()` to your template. It prints all "current" elements you can refer to.

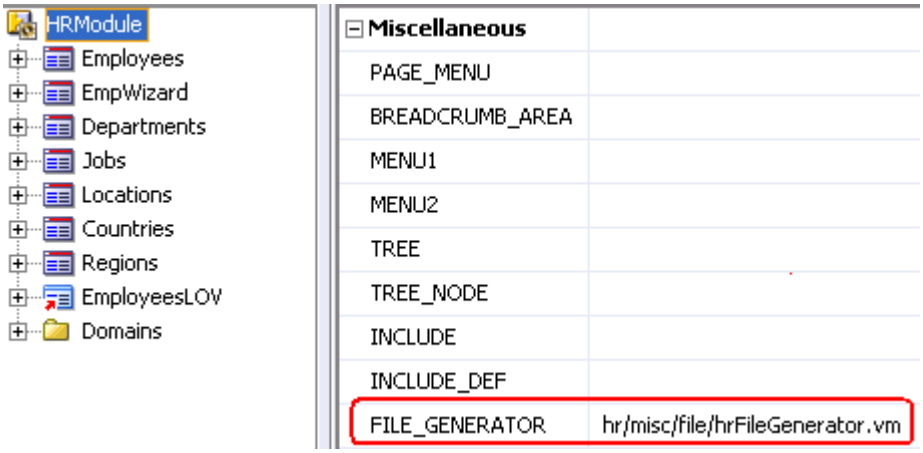
4.7.6. The File Generator Template

JHeadstart uses a special template, default/misc/file/fileGenerator.vm, as a means to generate additional files, not directly releated to a group. Examples of these files are the home page, ADF Faces region files referenced in generated pages to display header elements like branding images and global buttons, and SQL scripts to populate the JHeadstart database tables for table-driven features.

Each file that is generated through the fileGenerator.vm template has its own template that is used to generate the file content. The physical template path and name for each file is hardcoded in fileGenerator.vm.

```
## Generate log4j.properties
#set ($parsedContent = "#JHS_PARSE_NO_DEBUG('default/misc/file/log4jProperties.vm' ${JHS.service})")
$JHS.createFile("${JHS.sourceRootDir}/log4j.properties", $parsedContent)

## Generate Branding and Menu Global region files
#set ($parsedContent = "#JHS_PARSE_NO_DEBUG('default/misc/file/branding.vm' ${JHS.service})")
$JHS.createOrReplaceFile("${JHS.htmlRootDir}/common/regions/branding.jspx", $parsedContent)
#set ($parsedContent = "#JHS_PARSE_NO_DEBUG('default/misc/file/brandingAppContextual.vm' ${JHS.service})")
$JHS.createOrReplaceFile("${JHS.htmlRootDir}/common/regions/brandingAppContextual.jspx", $parsedContent)
#set ($parsedContent = "#JHS_PARSE_NO_DEBUG('default/misc/file/menuGlobal.vm' ${JHS.service})")
$JHS.createOrReplaceFile("${JHS.htmlRootDir}/common/regions/menuGlobal.jspx", $parsedContent)
```

As a result, configuring JHeadstart to use a custom template for a generated file is a bit different. Rather then going to the templates tab in the Application Definition Editor to change the name of the file template, you create a customized version of fileGenerator.vm and change the template name in this customized fileGenerator template. Configuring JHeadstart to use your custom fileGenerator template is done through the Templates tab at Service level in the Application Definition Editor.



By creating a custom version of the file generator template, you can also generate additional project-specific files using the JHeadstart metadata. The following methods on the JHS velocity context are available for generating files:

- **createFile**: this will generate a file when the file does not exist yet
- **createOrReplaceFile**: this will generate a file, possibly overriding an existing version of the file
- **createApplicationDefinition**: this will generate an application definition file when the file does not exist yet. The application definition file is registered as a special node in the JDeveloper Navigator to enable the Jheadstart context menu on the file.
- **createSQLScript**: this will generate a SQL script when the file does not exist yet. The generated script will be executed automatically when the service-level checkbox "Run Generated SQL Scripts" is checked (the default).
- **createOrReplaceSQLScript**: this will generate a SQL script possibly overriding an existing version of the file. The generated script will be executed automatically when the service-level checkbox "Run Generated SQL Scripts" is checked (the default).

Note that it only makes sense to create custom template for files that are re-generated even when the file already exists. Files that are generated only once, can be customized without loosing the changes.

4.7.7. Generating a JSF Navigation Rule from a Generator Template

By default, JHeadstart generates into the main faces-config file all navigation rules that are needed to navigate between your generated pages (for example by using tabs, child group buttons and details buttons). If you want to generate an extra (global) navigation rule for some reason, you can include it by calling **addNavigationCase** or **addGlobalNavigationCase** of the JHeadstart FacesConfigGenerator class.

For an example, see the default generator template for the Details button (default/button/detailsButton.vm):

```
…
action="${JHS.facesConfigGenerator.addNavigationCase (${JHS.page.name}, "details",
${JHS.current.pageComponent.detailsPage.name})}"
…
```

As you can see, the template for a navigation button can also cause the corresponding navigation case to be generated. The method addNavigationCase takes three parameters:

1. The "from" page
2. The outcome name
3. The "to" page

Similarly, addGlobalNavigationCase takes two parameters: just the outcome name and the "to" page.

Reference: See the Javadoc of `FacesConfigGenerator`, methods `addNavigationCase()` and `addGlobalNavigationCase()`.

4.7.8. Generating a JSF ManagedBean from a Generator Template

By default, JHeadstart generates into the group faces-config files all managed beans that are needed for the standard JHeadstart functionality. If you want to generate an extra managed bean for some reason, you can include it by calling **addCustomManagedBean** of the JHeadstart FacesConfigGenerator class.

For example you can put this in the generator template where you want to add the managed bean generation:

```
#set ($myCustomBean =
${JHS.facesConfigGenerator.addCustomManagedBean
    ( ${JHS.current.group},
      "custom/misc/facesconfig/myCustomBean.vm",
      "${JHS.current.group.name}MyCustomBean",
      ${JHS.current},
      ${JHS.page}
    )
})
```

This code generates the managed bean, and puts its name into a Velocity variable that you can use in the remainder of the generator template. For example, you can generate the name of the managed bean into a page by referencing `${myCustomBean}`.

The parameters of addCustomManagedBean are as follows:

1. The Group used to determine in which faces config file the bean should be added
2. The Velocity template file to use for this bean
3. Name of the bean
4. Current JHeadstart Model pointer
5. Current page

An example of the Velocity template for such a managed bean is:

```
<managed-bean>
  <managed-bean-name>
    ${JHS.current.managedBean.beanName}
  </managed-bean-name>
  <managed-bean-class>
    com.mycompany.myapp.controller.bean.MyCustomBean
  </managed-bean-class>
  <managed-bean-scope>
    session
  </managed-bean-scope>
</managed-bean>
```

Of course you must also create the managed bean class (in the example `com.mycompany.myapp.controller.bean.MyCustomBean`). You can then refer to the properties of the bean class in the properties of the ADF Faces components you generate into your pages. If for example the bean class has a method `getMyProperty()`, you can include the following in the generator template where you called addCustomManagedBean:

```
someAdfFacesProperty="#{${myCustomBean}.myProperty}"
```

Reference: See the Javadoc of `FacesConfigGenerator`, method `addCustomManagedBean()`.

Below we will give a few examples of customizations that illustrate the different ways you can customize JHeadstart generation. It also covers some customizations that occur often.

4.8. Generating Mobile Applications

ADF Faces makes it possible to run applications on mobile devices, like PDA or Telnet devices. JHeadstart leverages these ADF Mobile capabilities and adds its own features to simplify generating mobile applications.

JHeadstart support for Mobile devices is developed with certain main points in mind:

- As less as possible device specific code.

- Definition of capabilities of a device type are not hard coded, but stored in the jag-config.xml file.

This makes it possible to easily add additional device types to JHeadstart without code changes. Because the jag-config.xml file is in the project, you can add device types yourself and define the capabilities of a viewtype.

4.8.1. JHeadstart properties for Mobile

4.8.1.1. Service Level Properties

Following properties on the Service level are related to mobile support:

- The **ViewType ?** property defines the type of pages to be generated by JHeadstart. The generator uses this property to determine the set of templates to be used. JHeadstart by default is shipped with support for ADF Faces and ADF Mobile PDA.

- The **Use Short Labels ?** property is used to generate short label prompts. When this property is set, the generator uses the item property **Short Prompt** for generating the prompt in a page. Otherwise the normal prompts are used. Note: There is only one short prompt property, so JHeadstart makes no distinction for prompts generated in table layout and formlayout when using the short prompt.

- The **Generate JavaScript ?** property can be used to disable the generation of JavaScript in a generated pages. Useful when generating for a device that doesn't support JavaScript.

JHeadstart sets default values of these properties when creating a new Application Structure File. Defaults are taken from the defintion stored in the jag-config.xml

4.8.1.2. Group Level Properties

Following properties on the Group level are related to mobile support:

- The list of **Layout Styles** is restricted to layout styles supported for the selected View Type.

4.8.1.3. Item Level Properties

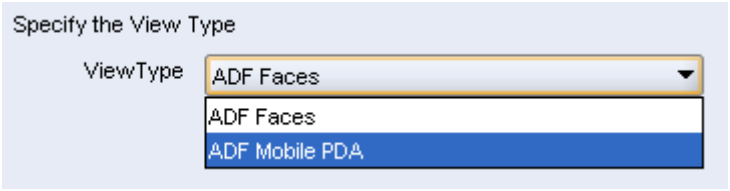
Following properties on the Item level are related to mobile support:

- The **Short Prompt** property can be used to define an alternative prompt for devices with less space. The generator uses this property when service level property **Use Short Lables** is set.

- The list of **Display Types** is restrictied to display types supported for the selected View Type.

4.8.2. New Application Definition Wizard

When creating a new Application Definition, you can choose the ViewType:



The service level properties **View Type ?**, **Use Short Labels**, and **Generate JavaScript ?** are set with default values for the chosen ViewType. Defaults are read from the definition in jag-config.xml.

The group level properties **Columns** and **Advanced Search Layout Columns** are set to the default value for the chosen ViewType.

4.8.3. JHeadstart Application Generator

For every ViewType, JHeadstart has a template binding file defined. JHeadstart uses this file to find the templates to be used for generating. The template bindings files are copied into your project in the Resources - config directory.

JHeadstart also supports a set of GeneratorText properties for each View Type. You can use this to generate short text for mobile devices. For example: 'Adv Srch' instead of 'Advanced Search'. The GeneratorText properties are copied into your project when you enable JHeadstart, so you can customize them from within your project. You find them in the Resources – nls directory.

The GeneratorText properties are used for generation of the NLS resources bundles for your application. The Generator text file used for generation depends on the selected ViewType. The relation between View Type and properties file used is in the jag-config file. See next section.

Attention: When reenabling JHeadstart for a project, these files are overwritten. Make sure to capture your changes before reenabling JHeadstart.

4.8.4. Customizing View Types

All properties of a ViewType are defined in the jag-config.xml. The JHeadstart Enable Project Wizard copies this file into your project.

The jag-config file defines beans used by JHeadstart. The following beans are related to mobile support:

- viewTypes Manager holds a collection of ViewType beans.
- viewTypeAdfFaces defines the properties of the ADF Faces View Type.
- ViewTypeAdfMobilePDA defines the properties of the PDA View Type.

You can change settings in these beans to change JHeadstart behaviour. It is recommended to not change the shipped definition viewTypeAdfFaces.

Attention: Be careful when changing the jag-config file. Incorrect settings here may (will) corrupt functionality of JHeadstart.

Attention: When reenabling JHeadstart for a project, this file is overwritten. Make sure to capture your changes before reenabling JHeadstart.

4.8.5. Adding a View Type

Take the following steps when you want to add a ViewType to JHeadstart:

1. Copy the definition of 'viewTypeAdfFaces'. Change the bean name and set properties as appropriate.
2. Add the new bean definition to the list of beans in viewTypeManager.
3. Optional, only when you need custom templates for the new ViewType: copy templatebindings.jtp to <yourtemplatebindings.jtp> Make sure that the file name for the template binding file matches the definition in jag-config.xml. Build the custom templates and put them in <yourtemplatebindings.jtp>

Optional, only when you want to change generatorText for the new ViewType: copy GeneratorText.properties for the language(s) you need and change them as you like. Make sure that the file name for the <your>GeneratorText matches the definition in jag-config.xml.

4.9. What was Generated for What Purpose

The table below describes the files you get in your project when generating with JHeadstart.

File Type / File	Location	Purpose
DataBindings.cpx	**View Package** property at service level	Container file for ADF Model layer.
*PageDef.xml	**Page Definitions Sub Package** property at service level	Page Definitions hold definition of ADF Data Bindings
JSF pages (*.jspx)	**UI Pages Directory** property at service level	JSF files define the application page using ADF Faces.
ADF Faces Regions (*.jspx)	**UI Page Regions Directory** property at service level	Region files are reusable pieces of ADF Faces pages.
ApplicationResources.properties or .java (or other name)	**NLS Resource Bundle** property at service level	Resource Bundles that contain language dependent texts and date(time) patterns.
faces-config.xml (or other name)	**Main Faces Config** property at service level	JSF Navigation Rules between pages.
JhsCommon-beans.xml (or other name)	**Common Beans Faces Config** property at service level	JSF managed beans that are common to every group in the Application Definition
[groupName]-beans.xml	**Group Beans Faces Config Directory** property at service level	JSF managed beans specific to groups in the Application Definition

CHAPTER

5 Generating Page Layouts

This chapter describes the various page layout styles JHeadstart can generate. The following layout styles are described:

- Creating Form Pages
- Creating Select-Form Pages
- Creating Table Pages
- Creating Table-Form Pages
- Creating Master-Detail Pages
- Creating Tree Layouts
- Creating Shuttle Layouts
- Creating Wizard Layouts

The last section of this chapter describes how you can change the overall page layout, like colors, fonts, margins, headers and footers.

5.1. Creating Form Pages

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

Make the following changes to your group in the Application Definition to generate a Form Page:

1. Set the **Layout Style** property to 'form'.

There are a couple of group properties in the Form Layout category that influence this layout style:

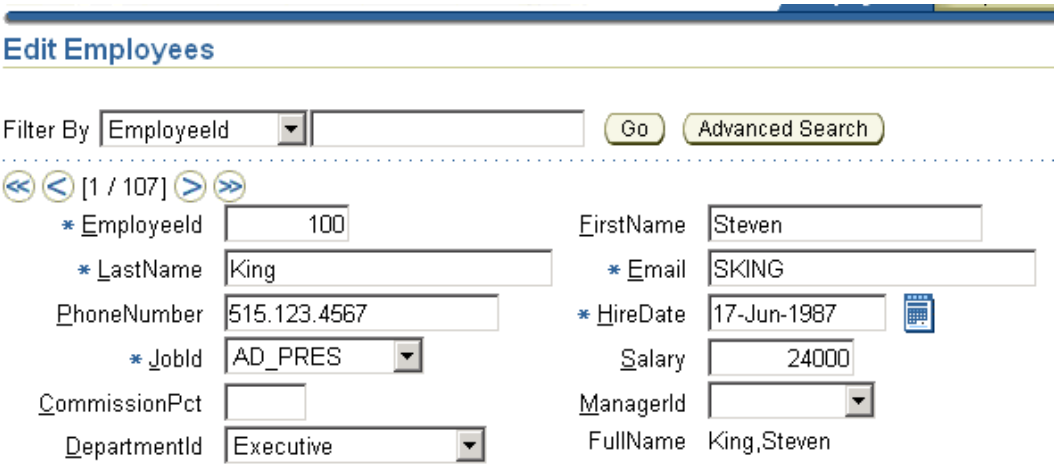
Form Layout	
Form Width	10%
Columns *	2

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 2, which will leave a page with all items placed below each other in one column.

Example of a form page:



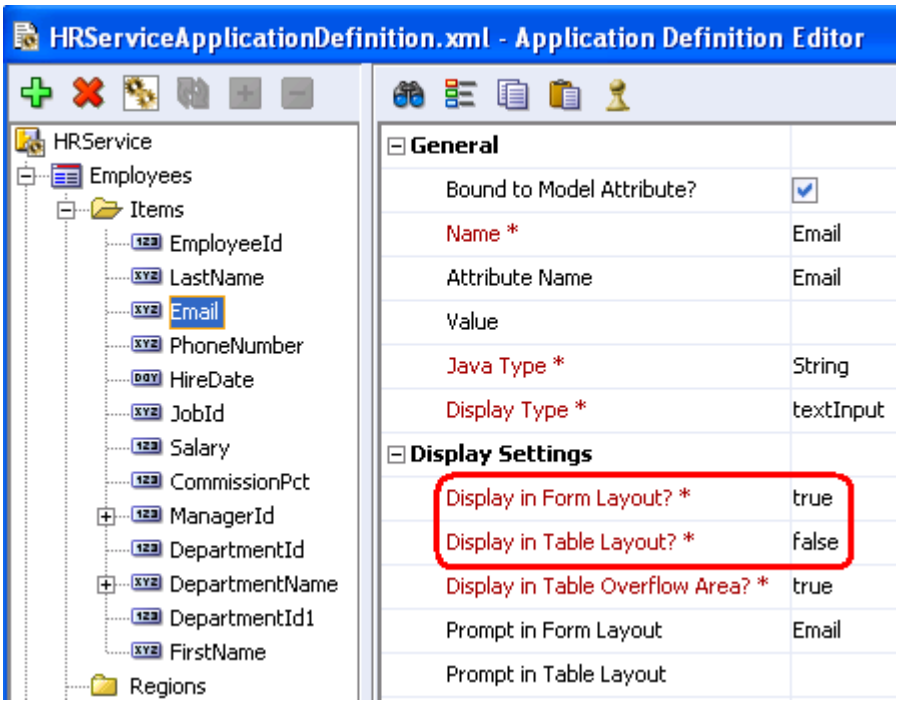
Notice that the items are laid out in two columns as specified in the **Columns** property.

Attention: The display sequence of the items on the generated page is determined by the order of the items in the group in the Application Definition. The items are layed out from left to right, and then to the next line.

Attention: The order of the rows when navigating to the next or previous record is determined by the Order By clause in the View Object. See section 3.3.5 - Determining the Order of Displayed Rows.

5.1.1. Hide Items on the Form Page

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



The **Display in Form Layout?** property has three possible values:

1. True. The item is generated in the form page
2. False. The item is not present in the generated form page
3. EL expression that is evaluated at runtime (must return true or false).

5.1.2. Using Regions

By default, JHeadstart places all items on the page in the order you have defined them in the group.

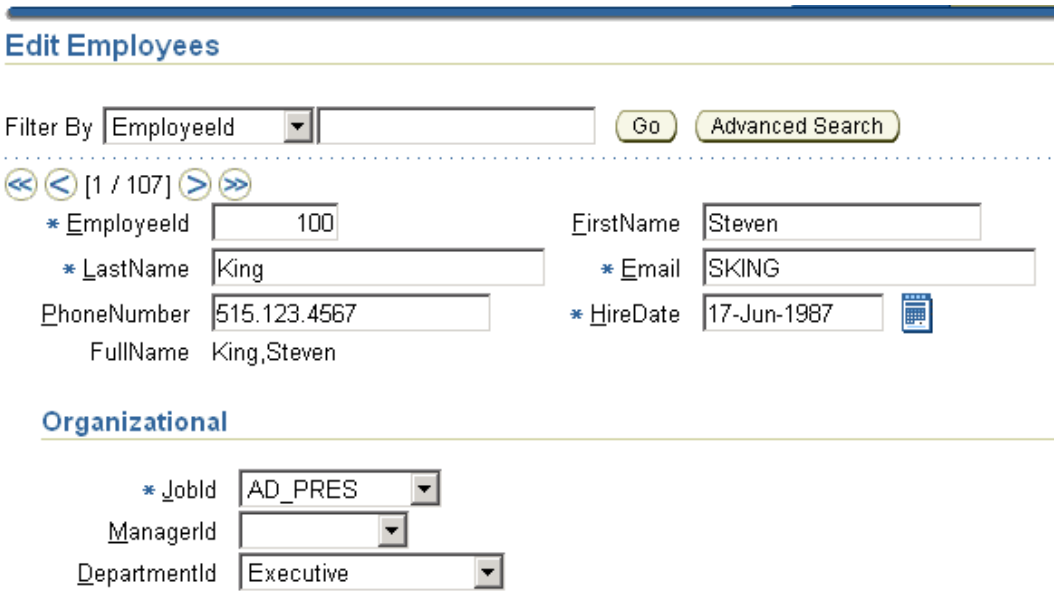
However, often you want to group related items together. For example: you have items with address information and you have items with financial information.

For this purpose, you can define Item Regions:

1. Within the group, add the regions you need. See section 4.4 - Using the Application Definition Editor, subsection 4.4.6 - Regions.
2. Move the items to the appropriate region.

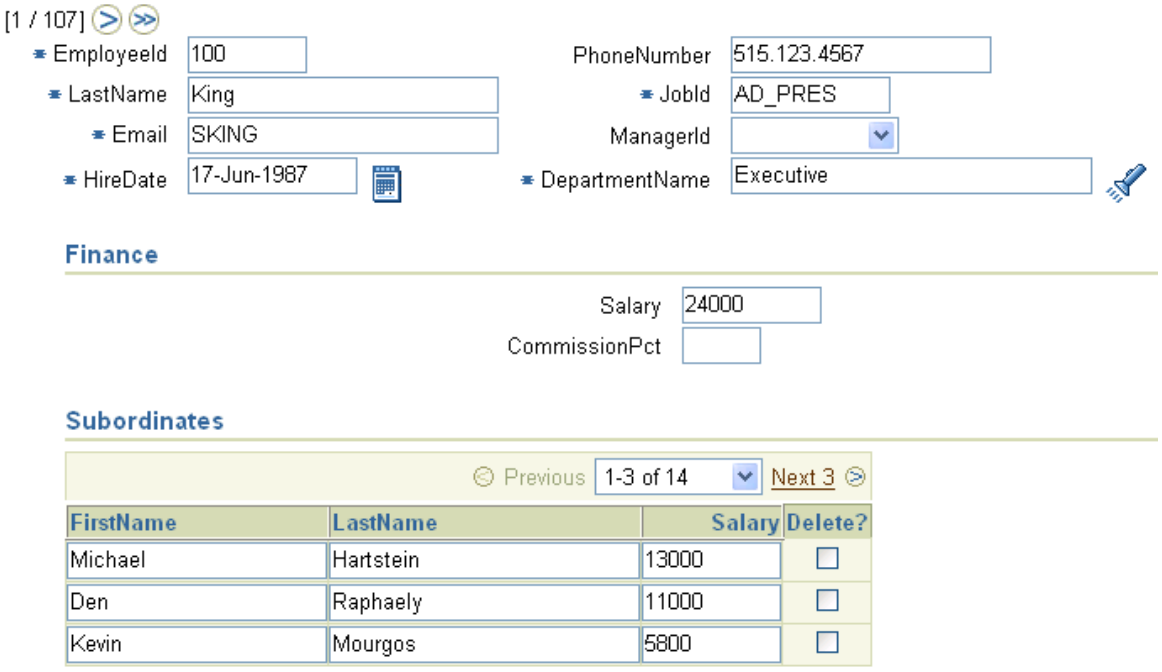
Attention: Regions are only used when generating single-row (form layout) pages or table overflow.

Example The screen shot below shows an Item Region titled 'Organizational'. The group level **Columns** property is set to 2. The region level **Columns** property is set to 1.

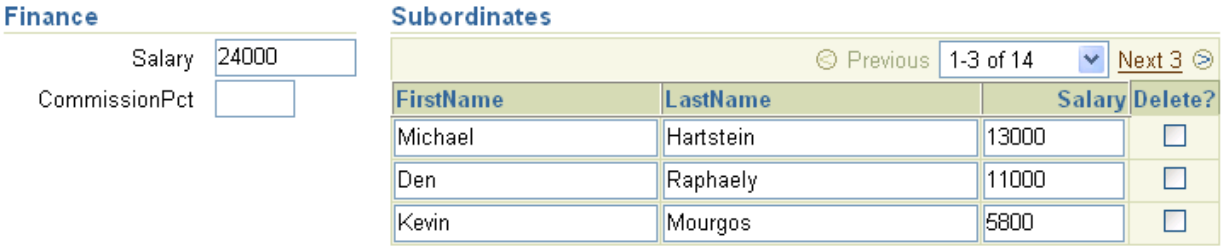


This is the simplest use of regions possible, for more advanced positioning of items Region Containers and Detail Group Regions can be used. A region container is simply a container for regions. Its most important property is the Layout Style. It determines how the regions are placed on the page.

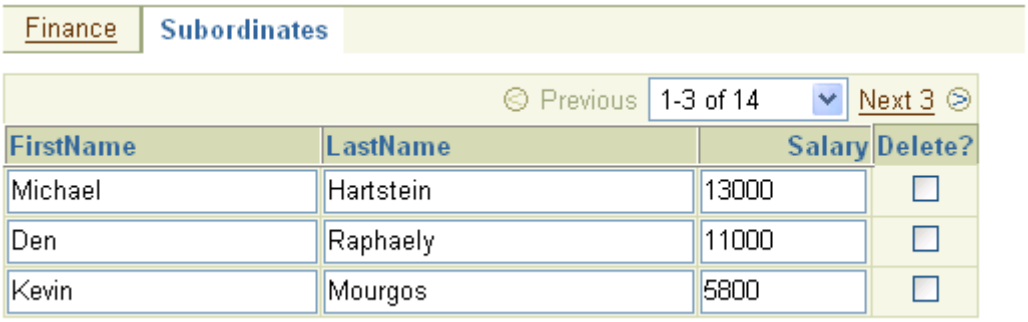
In the example below, the Layout Style of the Regions folder (which is also a Region Container) is set to 'vertical'. The Item Region 'Finance' and the Detail Group Region 'Subordinates' are layed out vertically.



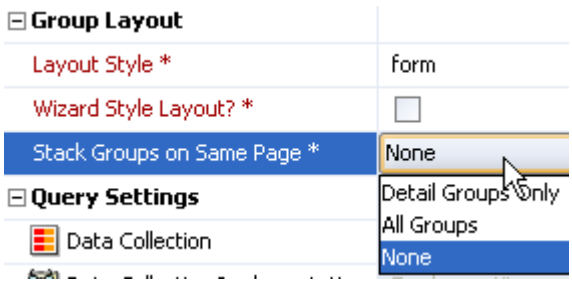
Region layout style 'horizontal' has the expected effect:



The most interesting option is the stacked one. The screenshot below shows an example of a region container with the stacked layout style. The regions are represented as tabs (hence the name stacked).

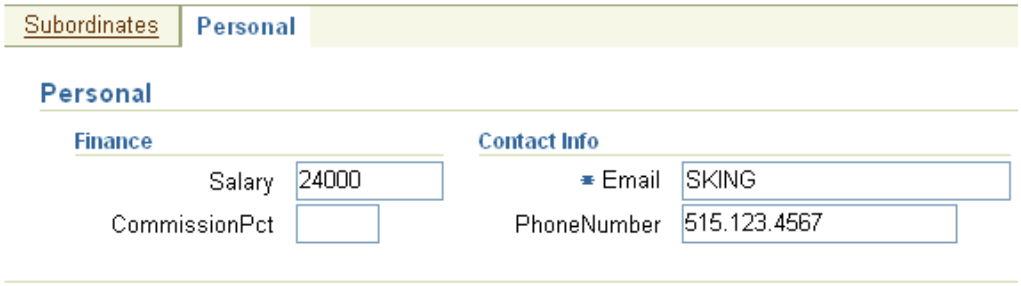


Suggestion: If the only regions you want to stack are Detail Group Regions, then you can use a shortcut: set the property **Stack Groups on Same Page** at group level to "Detail Groups Only". Then you don't need to create any regions. You can even stack the parent group itself, using the setting "All Groups". See also Section 5.5 - Creating Master-Detail Pages.



If you use nested Region Containers you can combine several layout styles.

Example `The Regions of a group consist of a Detail Group Region called 'Subordinates' and a Region Container called 'Personal'. The Region Container in turn contains 2 Item Regions called 'Finance' and 'Contact Info'.`

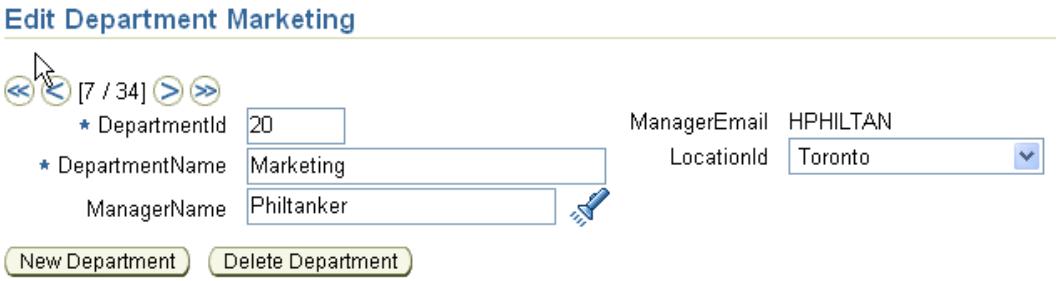


5.1.3. Create and Update Mode in Form Layout

JHeadstart does not generate separate pages for handling the creation of new records, or the update of existing records in for layout. 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. The verb used to prefix the **Display Title Singular** property (Enter or Create), is read from the templates/nls/GeneratorText resource bundle and can easily be changed. If you don't want a prefix at all, and just use the **Display Title Singular** property as is, then you can uncheck the group property checkbox **Add Verb to Form Title**. Note that this property is only visible in expert mode.
2. Components for record browsing are only shown in 'Update' mode.
3. New and Delete Buttons are only shown in 'Update' mode.

Page in 'Update' mode:



The same page in 'Create' mode:

When pressing a 'New…' button, the 'Create' action binding is executed and the `onCreate()` method in JhsPageLifecycle fires. This method stores an entry in the

managed bean Hashmap called `createModes`, with the name of the Create action binding as the key, and `Boolean.TRUE` as the value. As a result of this, the following expression can be used to check whether the Departments group is in create mode:

```
#{createModes.CreateDepartments}
```

Note that JHeadstart always suffixes the name of the Create binding with the group name, to prevent duplicate binding names when multiple groups are displayed on the same page.

With this knowledge you will understand the expression that is used to display the correct page title:

```
<af:panelPage title="#{createModes.CreateDepartments ?
nls['INSERT_TITLE_DEPARTMENTS'] :
nls['EDIT_TITLE_DEPARTMENT:#{bindings.DepartmentsDepartmentName}']}">
```

And the rendered property of the New and Delete buttons looks simply like this:

```
rendered="#{!createModes.CreateDepartment}"
```

When you press Save in the form page, the `onCommit()` in `JhsPageLifecycle` is called, and this method will remove all entries from the `createModes` managed bean Hashmap when commit is successful.

5.2. 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 Form page to enter or update a single row.

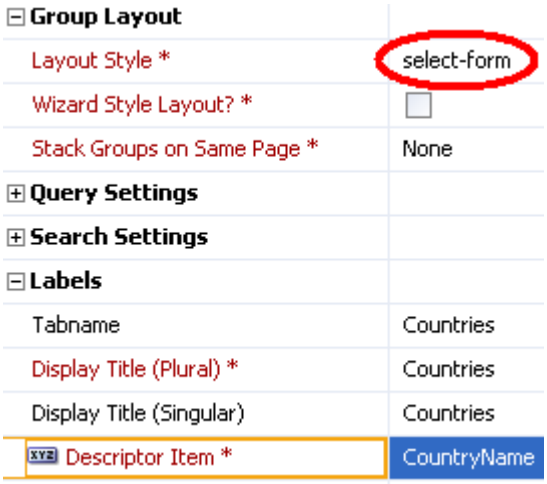
The Select Page contains a list box that displays one unique item from the group's data collection. The user can then find the appropriate row, select it, and perform the desired action (View, Edit, Delete, New). When New, View or Edit is pressed the Form Page is displayed. That page is similar to the page described in the section 5.1 Creating Form Pages above.

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

Make the following changes to your group in the Application Definition to generate a Select-Form layout:

1. Set the **Layout Style** property to 'select-form'.
2. Determine which item should be displayed on the Select Page. Specify that item to be the **Descriptor Item** of the group. You can only display one item in the list box, but it could be based on an attribute that contains a concatenation of a number of fields queried from the database. See section 3.3.6 "Create Calculated or Transient Attributes" on how to create such an attribute.
3. See the section 5.1 Creating Form Pages above to see which properties are appropriate for the Form Page. All properties that are appropriate to a Form layout also apply to the Form page of the Select-Form layout.

Example In this example the Countries group should be displayed in a select-form layout, and the item used for the Select Page should be CountryName.



Attention: The order of the rows in the Select Page is determined by the Order By clause in the View Object. See section 3.3.5 - Determining the Order of Displayed Rows.

5.3. 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 of a certain Department.

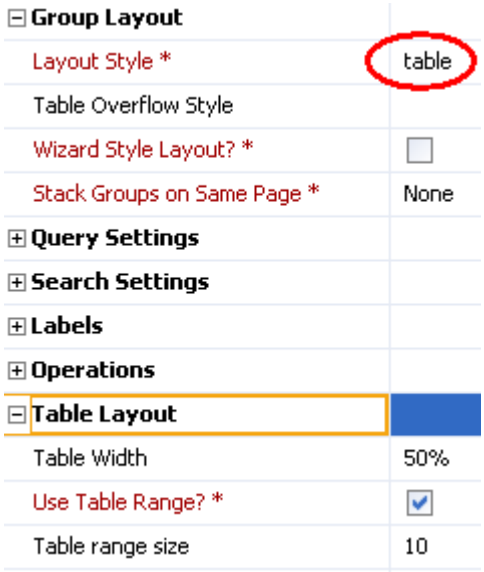
You can generate this type of page in a number of ways:

1. Using a Table Page. In a Table Page the data can be manipulated directly in the table. Use this option when the number of items in the group is small, so the table fits on the page. See the remainder of this section.
2. Using a Table Page with table overflow. Use this when the number of items is too large to fit on one row, and it is important to have all items in view with the entire table. With table overflow you can see some extra data for one of the rows on the same page as the table. See Section 5.3.5 Using Table Overflow.
3. Using a Table-Form Page. Use this when the number of items is too large to fit on one page, and you want a separate page with an overview of an entire row. This is a combination of a Table Layout with multiple rows and a Form Layout for manipulating one row. From the table page you can navigate to a form page to manipulate one row. See Section 5.4 Creating Table-Form Pages.

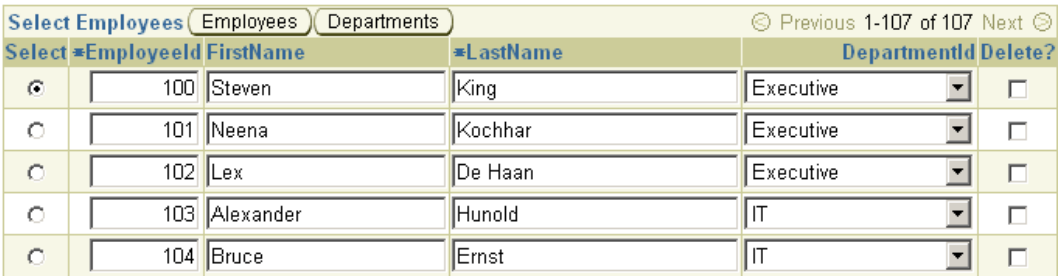
Make the following changes to your group in the Application Definition to generate a Table Page:

1. Set the **Layout Style** 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 `The Countries page should be displayed in a table layout. We make the following settings in the Application Definition Editor:`



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 section 3.3.5 - Determining the Order of Displayed Rows.

The remainder of this section discusses the following topics that are related to table layouts:

- Hide Items in a Table
- Allowing the User to Sort Data in a Table Page
- Limiting the Number of Rows on a Table Page
- Adding Summary Information to a Table
- Change Table-Related ADF Business Components Settings for performance tuning
- Using Table Overflow

5.3.1. Hide Items in a Table

With the **Display in Table Layout?** property, you can determine which items will be generated in the table. Values and meaning are the same as for the **Display in Form Layout?** property (see section 5.1.1 - Hide Items on the Form Page).

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

5.3.2. Allowing the User to Sort Data in a Table Page

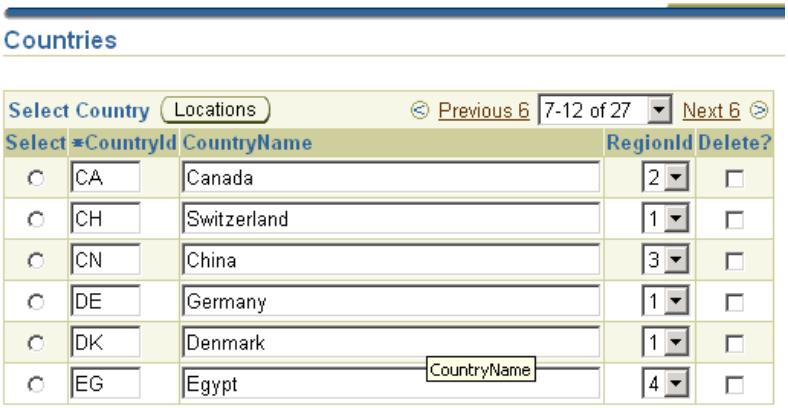
It is possible to generate a feature where the user can do an online sort of the records queried in a table. The user can simply click on the column header and then the table content is sorted based on the values in this column. Clicking the same column header twice will switch the sort order from ascending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

If this is required, then you must set the **Column Sortable** property on the item level to true, for those items you wish to be sortable.

5.3.3. Limiting the Number of Rows on a Table Page

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

1. Check the **Use Table Range?** property for the group in the Application Definition editor.
2. Set the **Table range size** property to the number of rows you want to display at once.



The above example shows the Countries group with the **Use Table Range?** property set to true, and the **Table range size** set to 6. You can now see that a dropdown list 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.

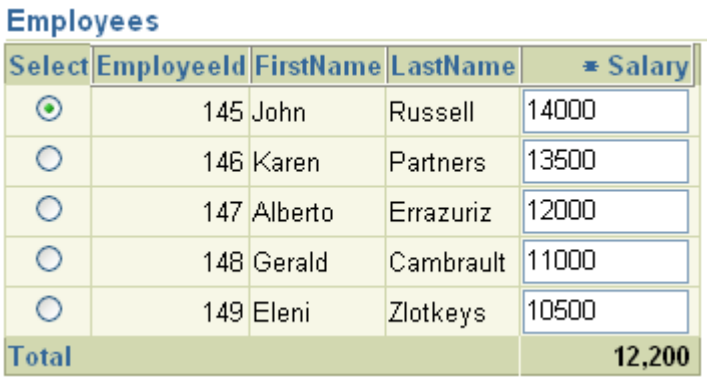
5.3.4. Adding Summary Information to a Table

For numeric columns, you can add summary information in the table footer. You can choose from 3 types:

1. Sum
2. Average
3. Count

The summary will be displayed in the table footer. The label of the table footer is by default Total, but you can change that in the Resource Bundle (see Chapter 11 "Internationalization and Messaging").

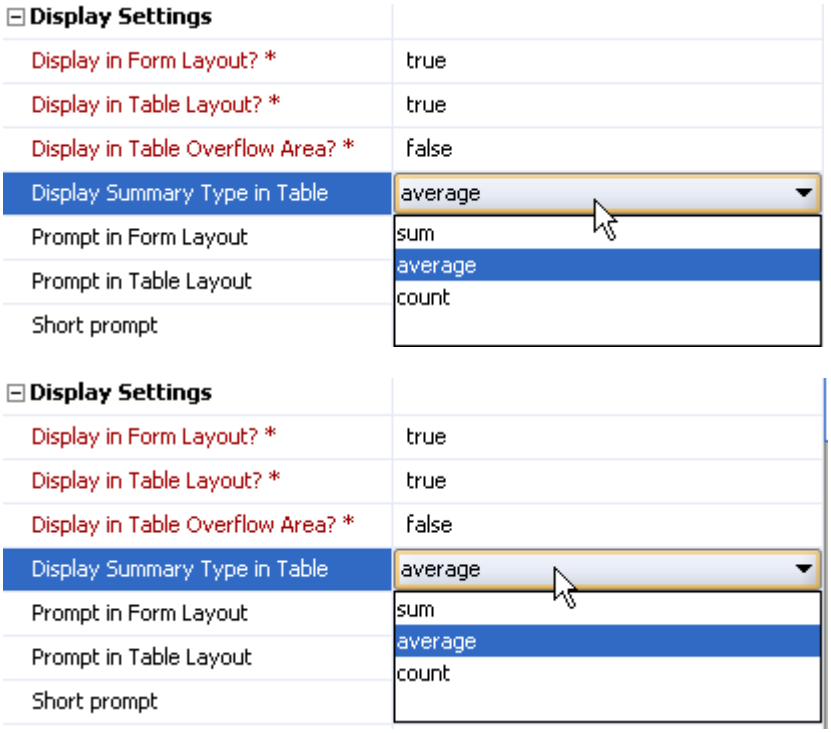
The example screenshot below shows the average salary in the table footer.



Employees

Select	EmployeeId	FirstName	LastName	* Salary
◉	145	John	Russell	14000
○	146	Karen	Partners	13500
○	147	Alberto	Errazuriz	12000
○	148	Gerald	Cambrault	11000
○	149	Eleni	Zlotkeys	10500
Total				12,200

To generate such a table summary, go to the item you want to summarize, and set the property **Display Summary Type in Table** to the desired type of summary.



Display Settings	
Display in Form Layout? *	true
Display in Table Layout? *	true
Display in Table Overflow Area? *	false
Display Summary Type in Table	average
Prompt in Form Layout	
Prompt in Table Layout	
Short prompt	

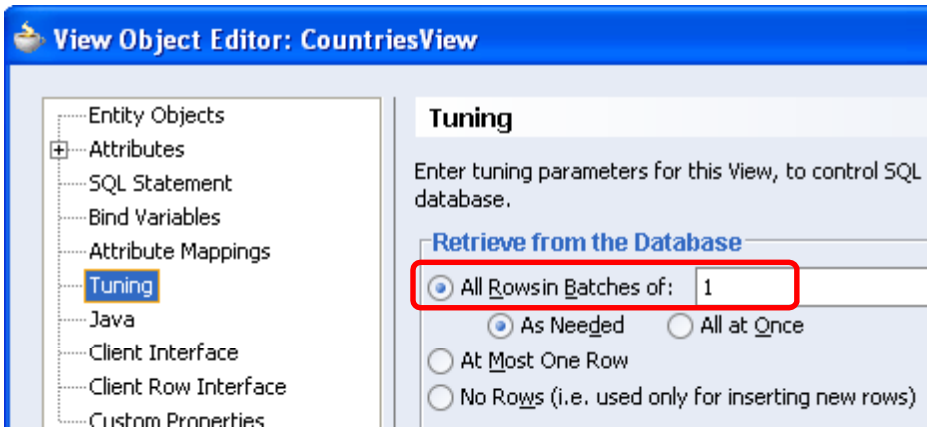
sum
average
count

Display Settings	
Display in Form Layout? *	true
Display in Table Layout? *	true
Display in Table Overflow Area? *	false
Display Summary Type in Table	average
Prompt in Form Layout	
Prompt in Table Layout	
Short prompt	

sum
average
count

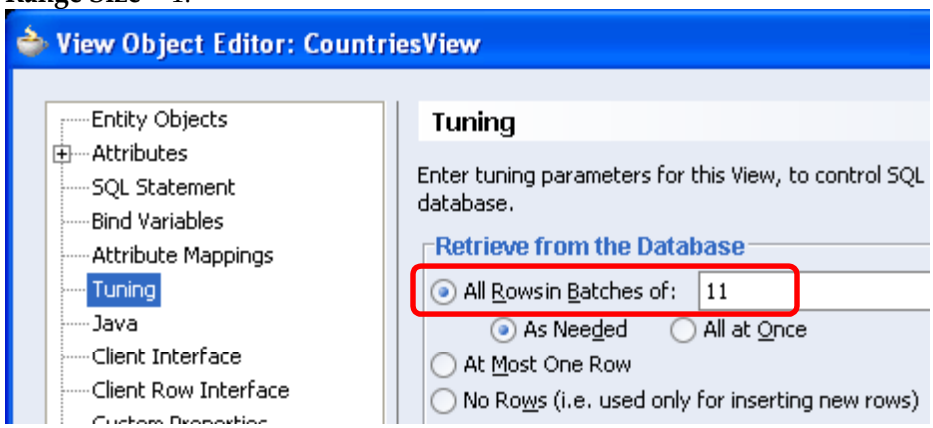
5.3.5. Change Table-Related ADF Business Components Settings

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.

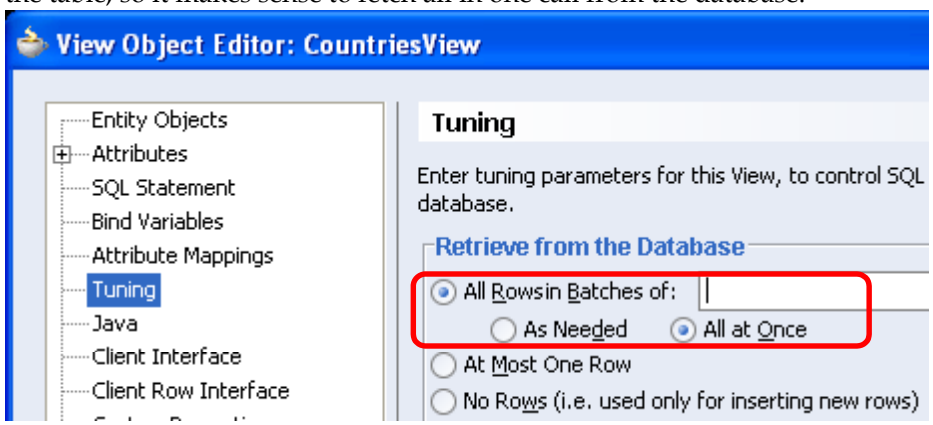


When retrieving multiple rows at a time, this is an unnecessary slow-down. So we recommend adapting the settings in the View Object to the settings in the JHeadstart group as follows:

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 or 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 makes sense to fetch all in one call from the database.

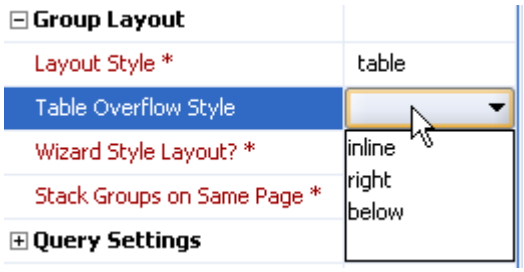


Reference: There is much more to say about ADF Business Components tuning. This sections explains only the ADF BC settings that are directly related to values of JHeadstart properties. For more information on ADF BC tuning, see the Oracle Application Developer Framework Developer's Guide for Forms/4GL Developers Release 10.1.3, section 27.2: Tuning Your View Objects for Best Performance (for example: section 27.2.4.2: Consider Whether Fetching One Row at a Time is Appropriate).

5.3.6. Using Table Overflow

It is also possible to include detail information for the current row of a table within the table page. This is called table overflow[1]. A table overflow area can be used when it is important to have the data in view with the entire table, but the user may or may not want to view the data at all times.

This can be achieved by using the **Table Overflow Style** property for the group.



1. When set to *inline*, the first column of the table is labeled "Details". Each cell in the column has an open/closed arrow icon followed by a link that reads "Show" or "Hide." When the user clicks Show, the overflow area is shown directly below the current row, and above the next row.

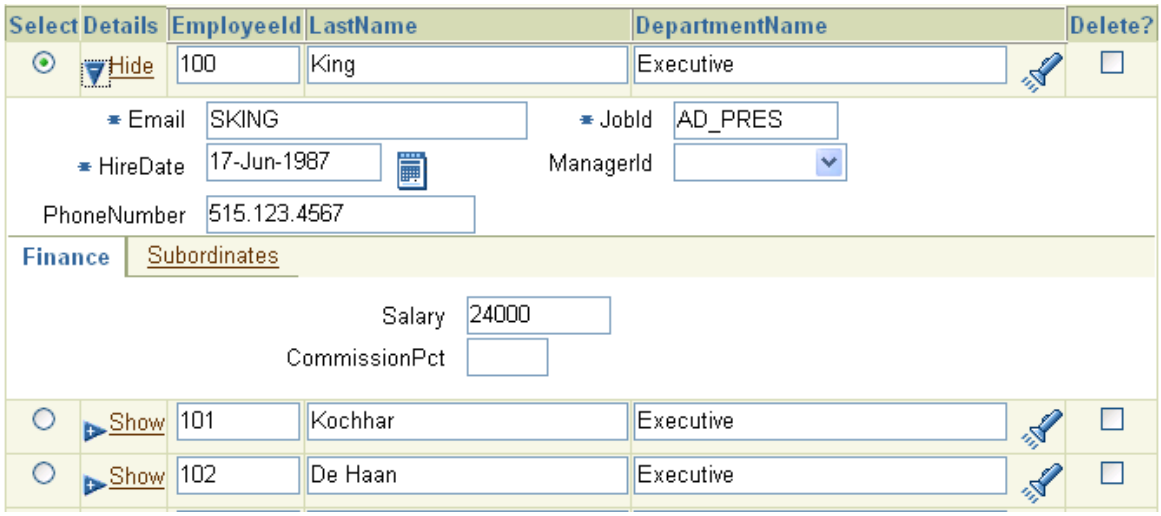
2. When set to *right*, the overflow area is always displayed. It is located to the right of the table.

3. When set to *below*, the overflow area is always shown. It is located below the table.

The overflow area displays all items that have the **Display in Table Overflow Area?** property set to true, plus any regions you may have defined if they contain items. It is also possible to show detail groups in the table overflow area, see Section 5.5 - Creating Master-Detail Pages.

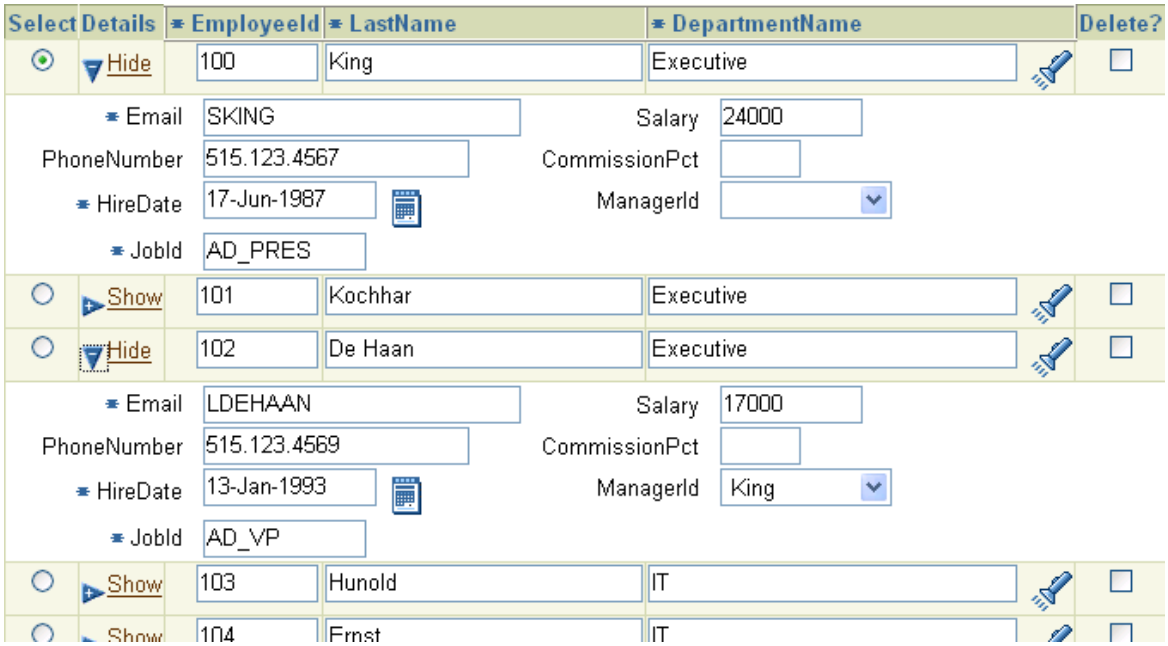
In the table overflow area you can apply regions similar to the way you use regions in form layouts, see Section 5.1.2 Using Regions. Below is an example of an inline table overflow style with stacked regions. The details are shown as inline data for a row.

[1] This feature was named Detail-Disclosure in JHeadstart 10.1.2 (UIX only)



Other options for the **Table Overflow Style** are below and right. As the names already suggest the details are put below or on the right of the base table. The most important advantage of using the inline style is the possibility to show several detail rows (instead of one) at the same time (see the Attention below). This is not possible when using the below or right inline table overflow style: only the details of the current row are shown.

Attention: By default only one inline table overflow area is shown at a time. If you open a second one, the first one closes. This was done because this behavior is required if you show detail groups in the table overflow, and by doing this always, the behavior is consistent.
If you want to be able to open more than one inline table overflow area at a time in cases where you don't have detail groups in the inline overflow, you can use a variation on the template `default\pageComponent\tableGroup.vm`. Comment out the else-branch of the if-statement that sets the disclosureListener, by putting ## in front of the else and the second disclosureListener line. See the comments in the template, and see Section 4.7 Using Generator Templates.



The above example shows the Employees group with the **Table Overflow Style** property set to inline. You can see that the inline overflow is displayed for both the first and the third record, which is only possible if you applied the customization described in the 'Attention' above. The table overflow area has two columns as defined by the **Columns** property for the group.

The following screenshot shows the same group, with the **Table Overflow Style** set to right. The details have been split into two item regions (salary and other). The **Layout Style** of the Regions folder is vertical.

5.4. Creating Table-Form Pages

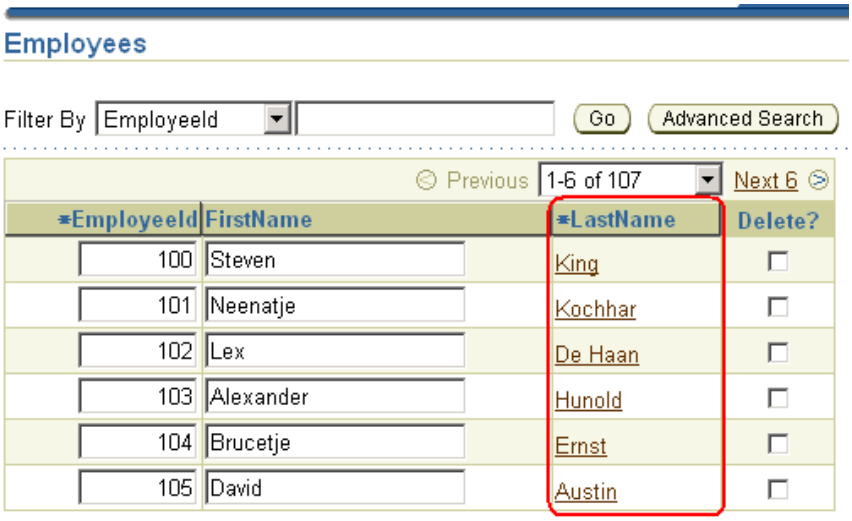
A Table-Form page is a combination of a multi-row page called a Table Page and a single row page called a Form Page. In the Table Page the user can update and select a row. If the user selects a row in the Table Page and presses the button or hyperlink to view the details, then the Form Page opens, and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

Attention: The Table-Form page layout consists of a combination of the Table layout and the Form layout. You can use the group properties described specifically for Table layout to layout the Table part of the Table-Form layout. Similarly, 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 (5.3) and Form pages (5.1) to see which properties are available.

Steps to create a table-form page:

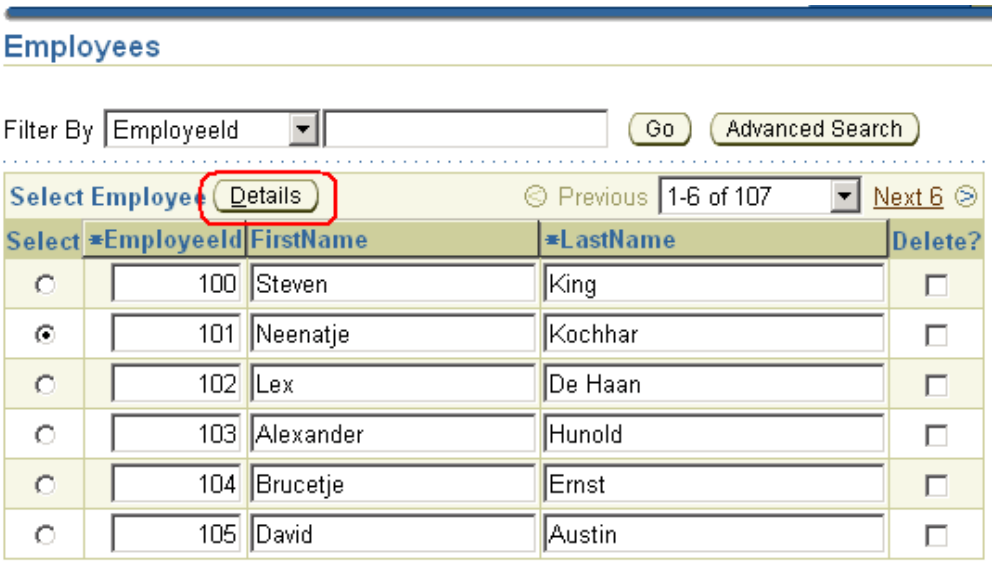
1. Set the **Layout Style** property to 'table-form' to generate a Table-Form Page.
2. Set **Display in Table Layout?** property to true for items you want to have in the table page.
3. Set **Display in Form Layout?** property to true for items you want to have in the form page.
4. Choose between a button or hyperlink for the means of navigation to the form page by setting the **Table-Form link** property for the group.

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



Suggestion: The link is generated on the descriptor item. It is therefore a good idea to set the descriptor to a unique key to help the user distinguish between the rows and make the descriptor item the first item of the group.

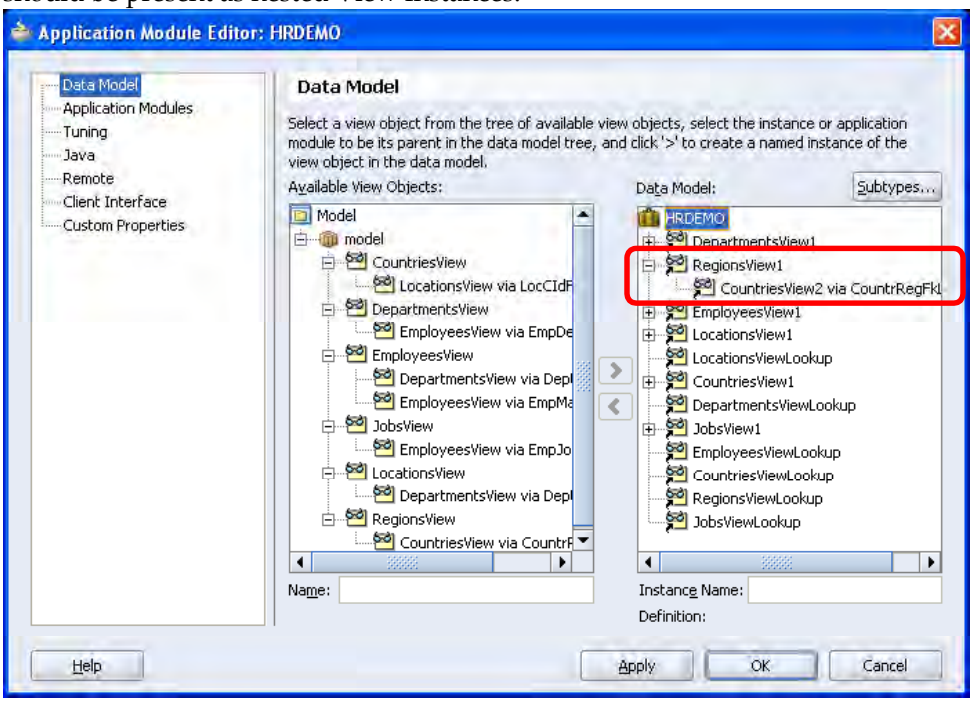
When you choose a button for navigation to the form page, you will get this:



5.5. Creating Master-Detail Pages

You may want to create pages that are related together as in a master-detail (parent-child) relationship. To be able to generate such master-detail 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 Section 3.3.7 - Setting Up Master-Detail Synchronization.

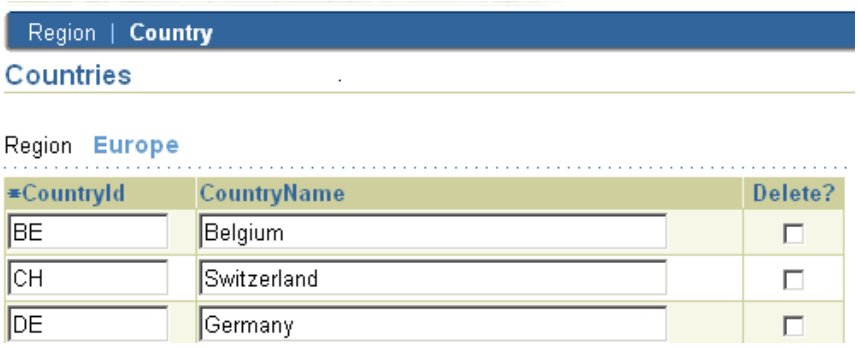
2. Create a group in the Application Definition for the master page, and create a detail group for the detail page. Set the Data Collection of the master group to the master View Object and the Data Collection of the detail group to the detail View Object. Note that for the detail group, you can only 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.

5.5.1. Master-Detail on Separate Page

1. Uncheck the **Same Page?** property 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. The level 2 menu bar will contain a tab for the master group (determined by its Tabname property) and one for the detail group. Determine the name for the level 2 tab of the detail. Specify this by using the **Tabname** property for the detail group.

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

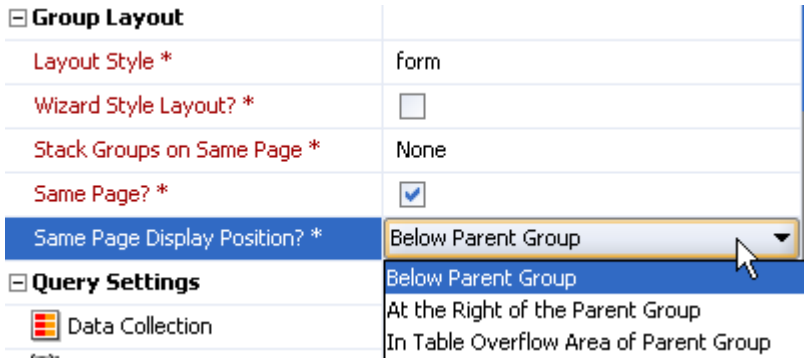
`As you can see two subpages are generated on the Regions tab. The name of the tabs are as defined by the` **`Tabname`** `property. When pressing the Country tab we get the second page:`



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 Item** specified for the master group.

5.5.2. Master-Detail on Same Page

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

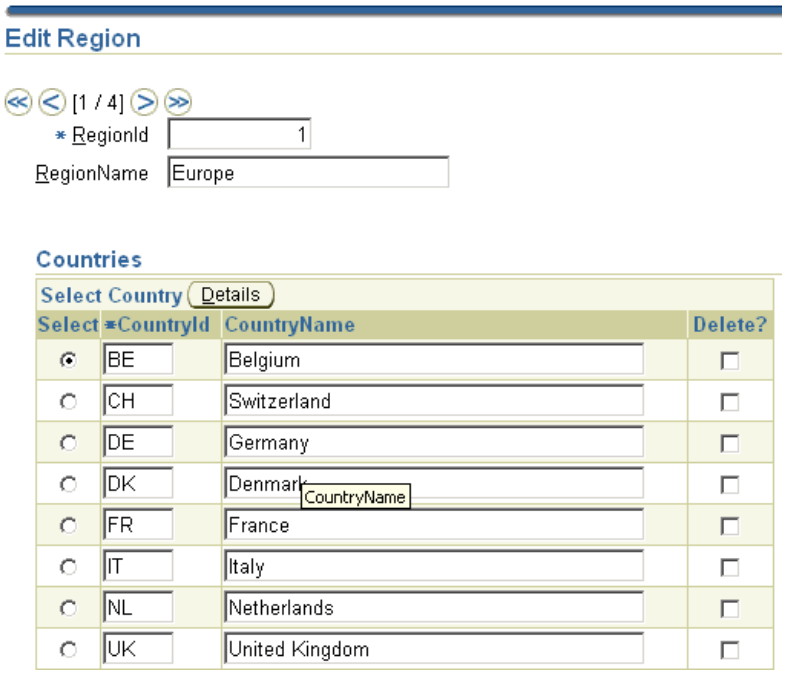


1. Check the **Same Page?** property for the child group. This indicates that the detail should be generated on the same page as its parent.
2. Set the **Same Page Position** property to the desired value: "Below Parent Group", "At the Right of the Parent Group", or "In Table Overflow Area of Parent Group".

3. Determine the header of the child group as you want it to be displayed above the child group. Specify this by using the **Display Title (Plural)** property for the detail group.

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

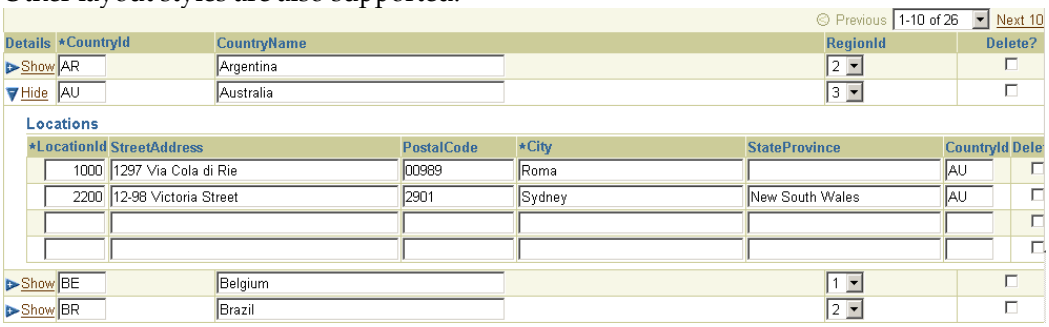
As you can see the master and the detail group have been generated on the same page in a master-detail layout. Note that the header above the details has been set as defined by the **Display Title (Plural)** property. The header above the base group Regions is determined by the **Display Title (Singular)** property (of the base group).



5.5.2.1. Show Nested Table

Using the **Table Overflow Style** property for the group, you can generate detail group (or child) tables within the table overflow area.

For this to work, the **Layout Style** of the parent group must be set to "table" (with layout style "table-form" the detail table will only be shown on the form page), the **Table Overflow Style** of the parent must have a value (for example inline as in the screenshot), the detail group must have the **Same Page?** checkbox checked, and the **Same Page Position** property of the detail group must be "In Table Overflow Area of Parent Group". The **Layout Style** of the detail group does not have to be table (as in the screenshot). Other layout styles are also supported.



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

5.5.2.2. Stack Groups on Same Page

If a master group has several detail groups that need to be displayed on the same page, a common design is to stack the detail groups, so that each has its own "tab". You can achieve this using Detail Group Regions, but a quicker way is to set the parent group property Stack Groups on Same Page to the desired value. For more information, see Section 5.1.2 Using Regions.

5.5.2.3. Combining Layout Styles on Same Page

Using Region Containers, Detail Group Regions and Item Regions, you can create advanced designs for your master-detail pages. For more information, see Section 5.1.2 Using Regions.

5.6. Creating Tree Layouts

You can use JHeadstart to generate tree controls. A tree control is extremely useful for showing hierarchical structures in your data model.

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.

5.6.1. Generating a Basic Tree

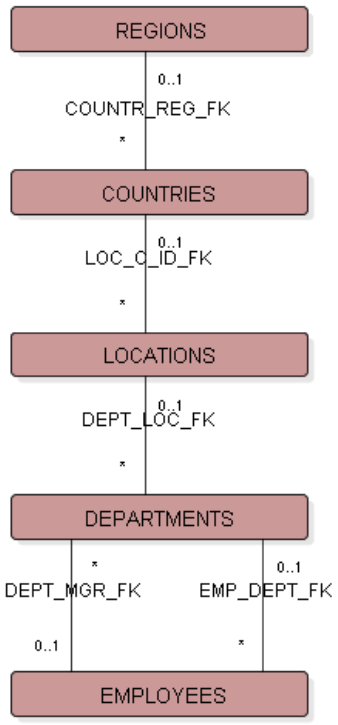
Most of the tree controls you will generate will be of the basic category. There are a few variations that will be explained in later sections:

- 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

It is advised to start with the basic steps, before reading the variations.

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.

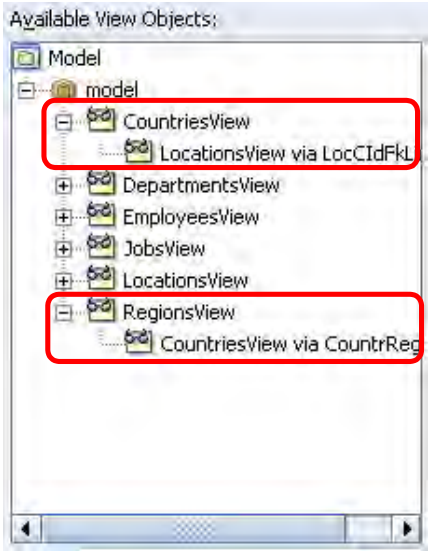
So this is our database diagram:



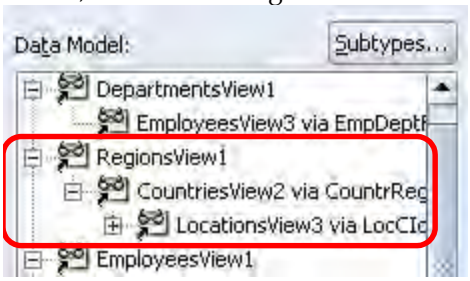
We will generate a tree with REGIONS, COUNTRIES and LOCATIONS.

Steps:

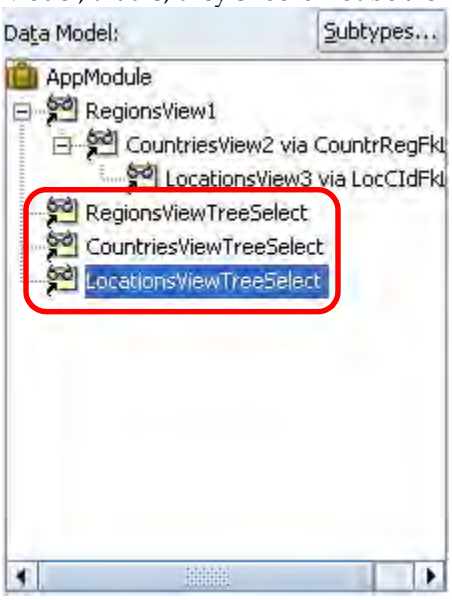
1. Check your model. Make sure you have a View Object for each level you want to show in the tree control. Check the presence of View Links between the View Objects. In this example, you will need a View Object for REGIONS, for COUNTRIES and for LOCATIONS and two View Links for the foreign keys between 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 Data Collections (also known as View Object instances or VO 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 Data Collection in the Data Model, and click the right arrow button. You should end up with something like this:

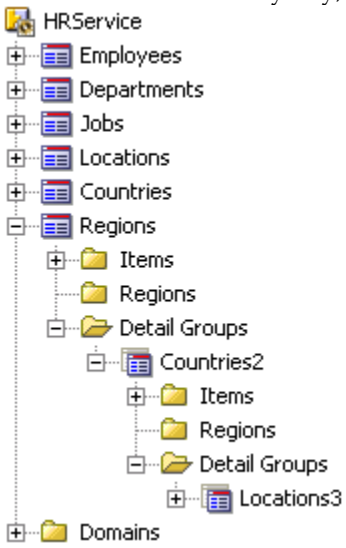


3. For selecting nodes in the tree, you need an extra Data Collection for each type of tree node. It is recommended to create dedicated Data Collections for tree selection, for example RegionsViewTreeSelect, CountriesViewTreeSelect, and LocationsViewTreeSelect. These usages should be top-level usages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



4. Make sure the JHeadstart Application Definition has groups and detail groups for your tree. You can maintain your groups by hand, or you run the 'New Application

Definition' wizard. Anyway, you should have something like this:



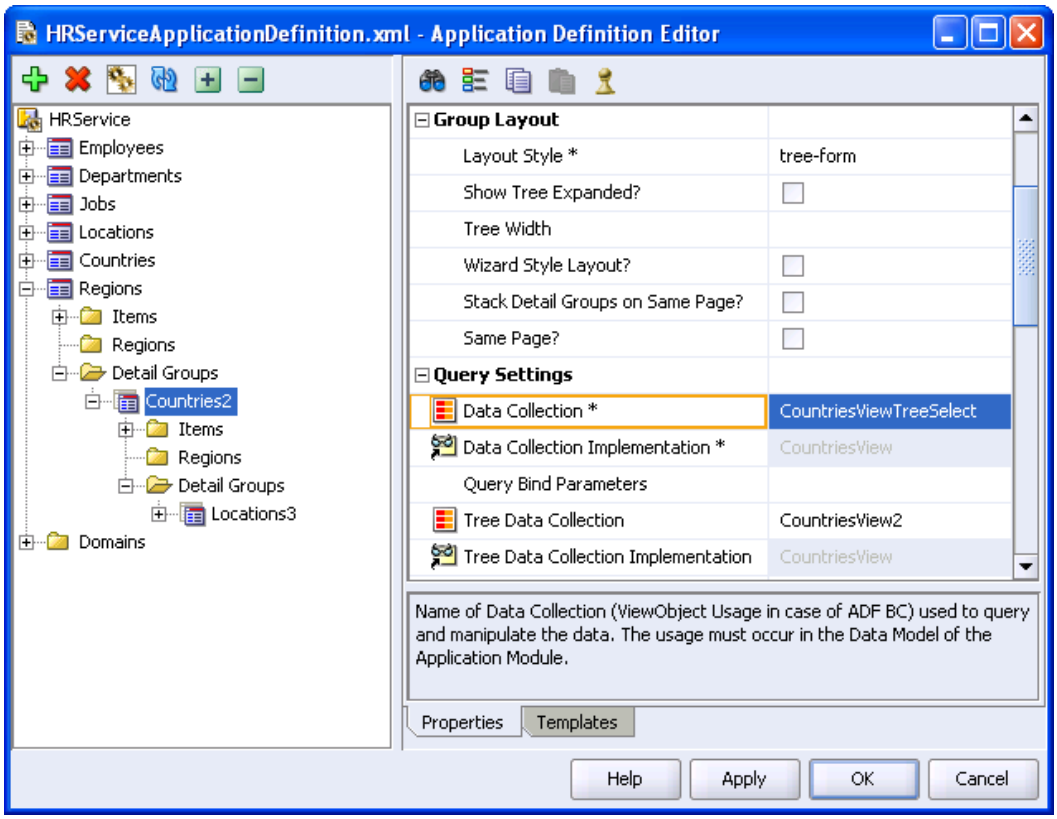
5. For the Regions group and all detail groups below it, change the **Layout Style** property to 'tree-form'. The layout style 'tree' will be discussed in one of the next sections.

6. For each of the tree groups, we need to specify two Data Collections:

   The **Data Collection** property specifies the usage for selecting a tree node and viewing / maintaining the data in a form layout. This should be a separate TreeSelect Data Collection at top level in the Application Module, or nested below the same Data Collection as the tree top-level group is nested below (if any).

   The **Tree Data Collection** property specifies the usage for showing the hierarchical structure of the tree, and needs to be a child usage of the direct parent group's Tree Data Collection. If there is no parent group, the Tree Data Collection must be a top-level Data Collection in the Application Module.

   If you originally created the groups using the New Application Definition Wizard, change the **Tree Data Collection** to be the same as the generated **Data Collection** property, and then change the **Data Collection** property to the TreeSelect usage of that view (for example RegionsViewTreeSelect, CountriesViewTreeSelect, etc) in all tree detail groups. Only in the tree top-level group, the **Data Collection** and **Tree Data Collection** can have the same value. When you want to use Quick Search or Advanced Search on the tree top-level group, the two properties must have the same value.

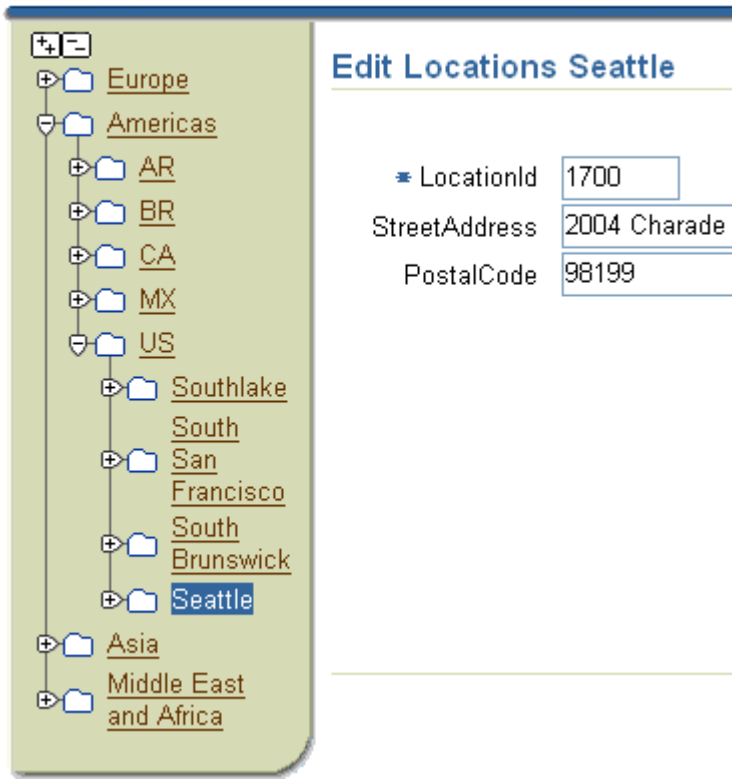


Attention: The reason why a separate TreeSelect Data Collection is required here is the following. If you select a child node in the tree, and you would use the Tree Data Collection to set the current row for display in the form area, the row might not be found because the current row in the parent View Object instance might be different from the parent node in the tree.

7. Select the correct **Descriptor Item** for each group to determine which item 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 the basis for the descriptor item. See section 3.3.6 "Create Calculated or Transient Attributes" on how to create such an attribute.

8. By default, the tree is rendered in collapsed mode 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 checking the property **Show Tree Expanded?** for the top-level tree group.

9. Change the **Tree Width** property and/or the Form Layout properties to your liking and run the JHeadstart Application Generator. You will get something like this:



Attention: There is a known issue in the ADF Binding layer (5149012) that causes leaf nodes to show up as expandable with the +-sign. Even after expanding a leaf node, the collapse sign is still displayed. This will be fixed in a future JDeveloper release.

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.

5.6.2. Variation: Basic Tree with navigation-only nodes

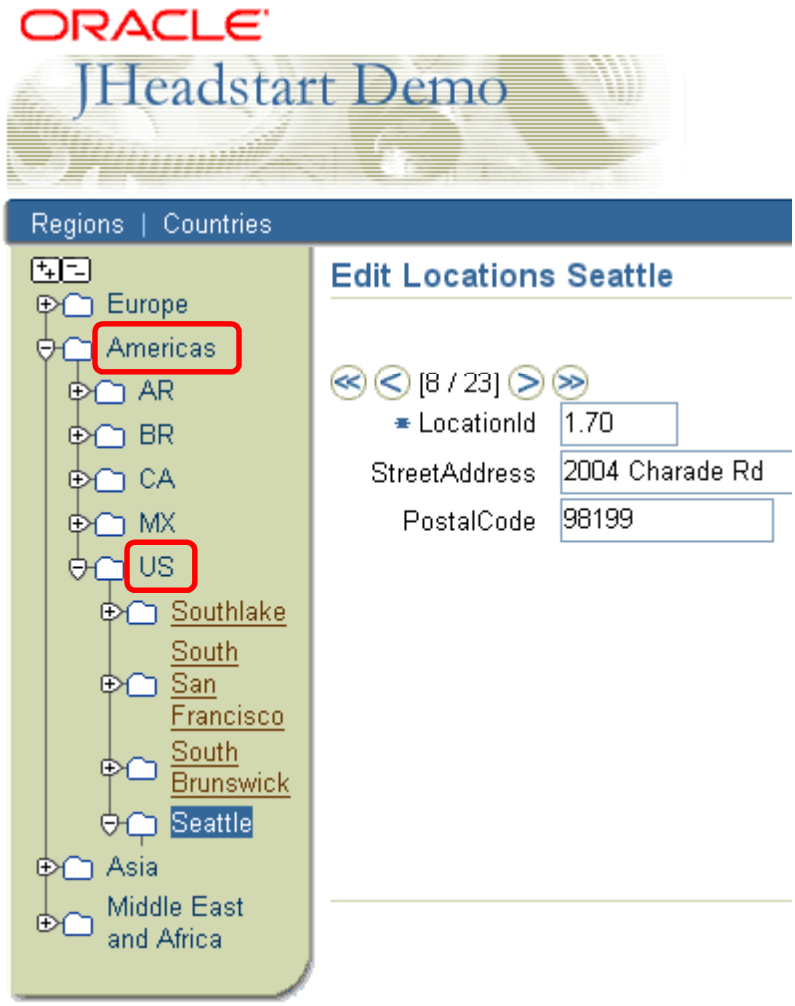
Suppose you do not need editing capability on each level of your tree. For example, you need REGION and COUNTRY only to drill down to the desired LOCATION. In this case, you change the **Layout Style** property of some of the groups to 'tree'. With this layout style, JHeadstart will use the group as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

These steps assumed you already applied the steps in section 5.6.1 Generating a Basic Tree.

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

Attention: The value of the **Data Collection** (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.

5.6.3. Variation: Recursive Tree

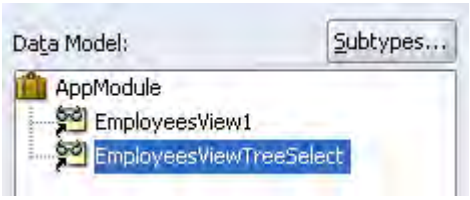
In some cases, tree structures are modeled in the database with self-referencing foreign keys (visible in Entity-Relationship diagrams by the so-called pig's ear).

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

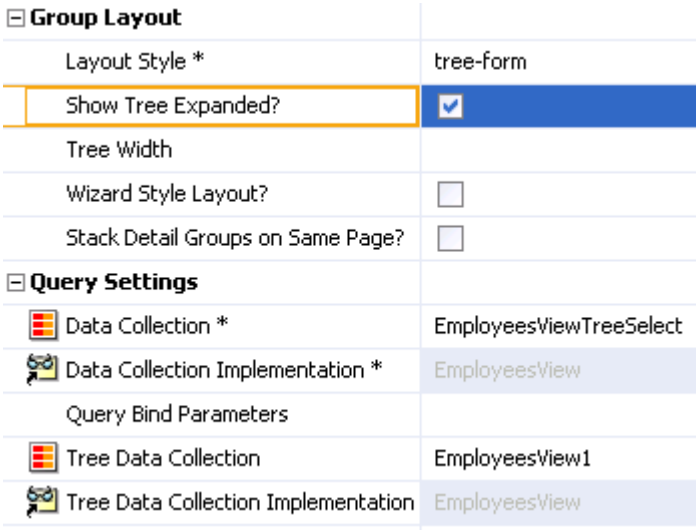
Generating a tree for such a situation is only slightly different. The steps of section 5.6.1 Generating a Basic Tree are applicable to this situation, though with minor changes. The step numbers here are variations of the basic tree steps with the same number.

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 a View 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 Definition 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 Data Collection** to be the same as the original Data Collection property (this usage is for showing the hierarchy of tree nodes), and change the **Data Collection** property to the Tree Selection usage of that view (for example EmployeesViewTreeSelect).
7. Select the right **Descriptor Item**.



8. If desired set the **Show Tree Expanded?** property

9. Change the **Tree Width** property and/or 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 can see, the tree can expand to any level.

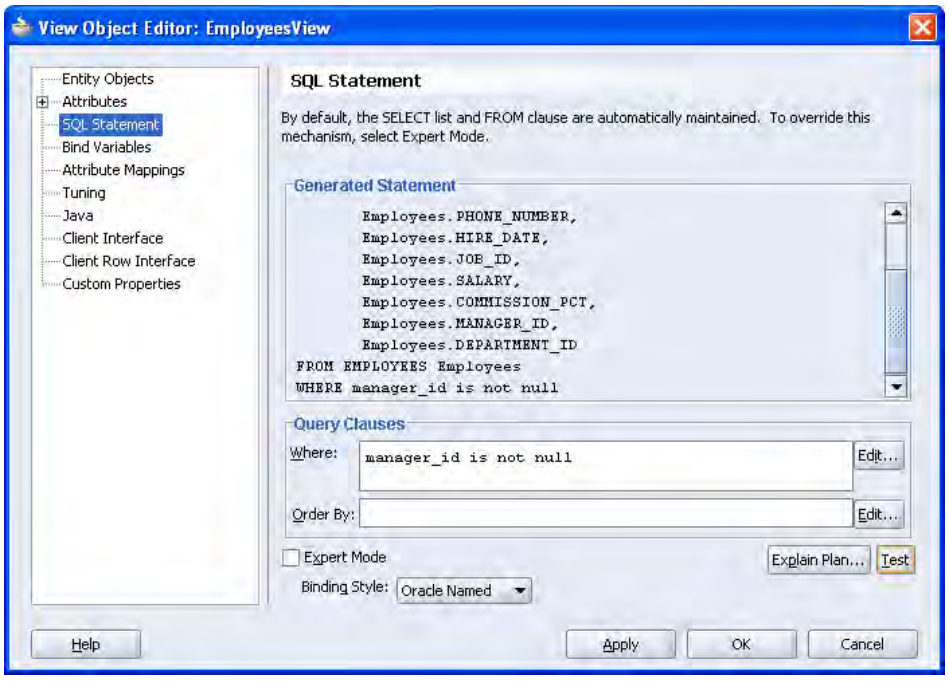
5.6.4. Variation: Recursive Tree with Limited Set of Root Nodes

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 5.6.3 Variation: Recursive Tree.

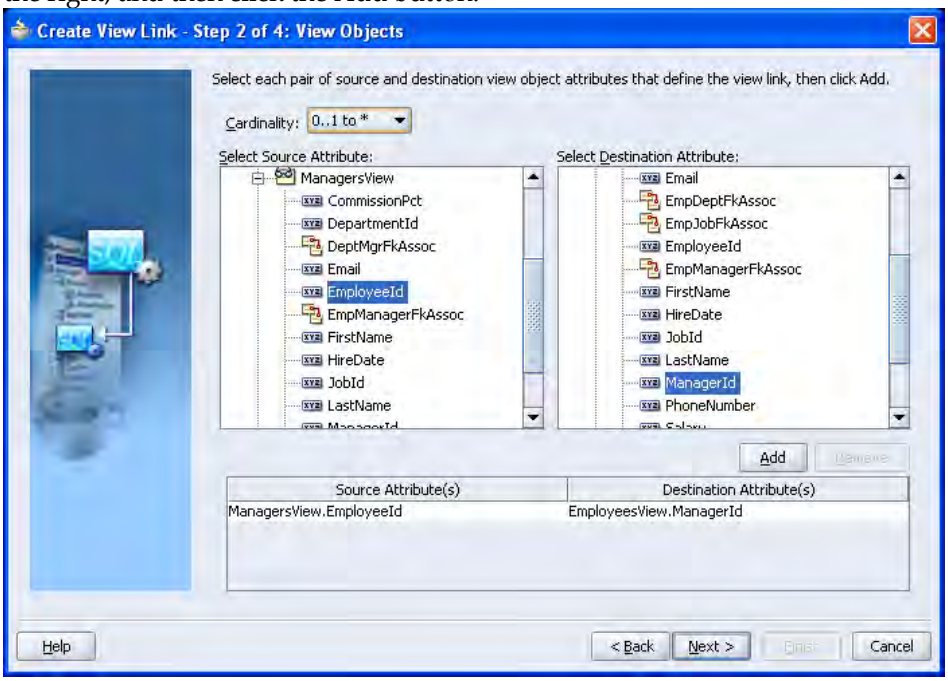
- Go to your Business Components project and choose 'New View Object'.
- Give the View Object a nice name ('ManagersView') and select the Entity Object and all its attributes.

- 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 (model) project, choose 'New View Link'.
- Give the View Link an appropriate name, for example MgrHasSubordinatesLink.

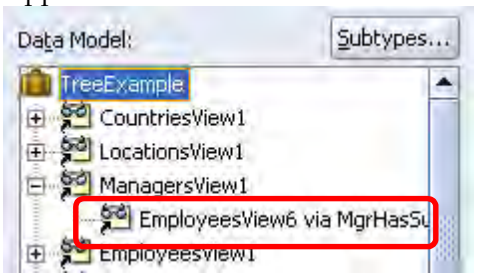
- In step 2 of the 'Create View Link' wizard, select the attributes that relate the data in the two views. In the case of managers and employees, you will select ManagersView.EmployeeId on the left, and EmployeesView.ManagerId on the right, and then click the Add button:



- 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.

- Change the value of the **Tree Data Collection** and **Data Collection** property of the Employees group to the new Data Collection 'ManagersView1'.
- Set the **Recursive Tree Data Collection** of the Employees group to the child Employees usage of ManagersView1. This information can be found in the Application Module Data Model.



- 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.

5.6.5. 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 5.6.3 Variation: Recursive Tree. We cannot use 5.6.4 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.

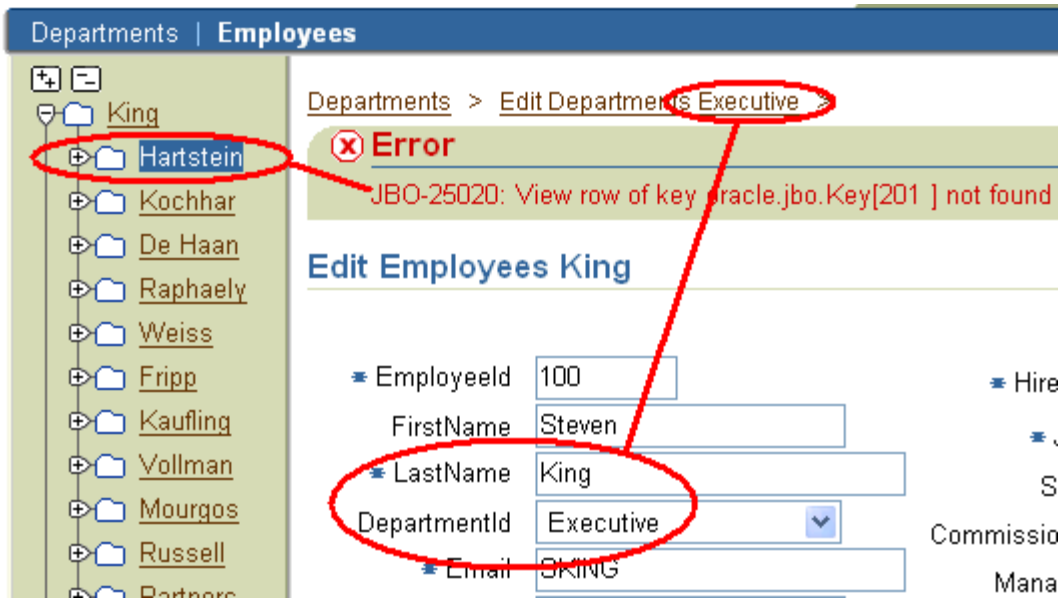
- First ensure that the Application Module's data model includes the parent-child relations we want to use. We should have a top level DepartmentsView usage, with an EmployeesView child.
- In the Application Definition Editor, create a new Base Group and call it Departments. Set the **Data Collection** to the top level DepartmentsView usage. Enter a Tab Name, Display Titles, and a Descriptor Item.
- Set Advanced Search to samePage and Quick Search to singleSearchField, on item DepartmentName.
- Make the Employees group of the recursive tree a detail group of the newly created Departments group. Save, leave and reopen the Application Definition Editor to reset the dropdown lists.
- Change the **Tree Data Collection** of the Employees child group to the child usage of the Departments group.
- Change the **Data Collection** 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 Data Collection 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 Data Collection 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 'Marketing'), 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 was clicked) belongs to the right department. This is because we set the Data Collection 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, which is not an employee of the Marketing 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 Hartstein in the tree, we still see employee 'King' in the Edit page. There is an error message ' JBO-25020: View row of key oracle.jbo.Key[201] not found in EmployeesIterator'. Of course, the EmployeesView3 usage now includes only the employees of Human Resources, which does not include Hartstein!



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.

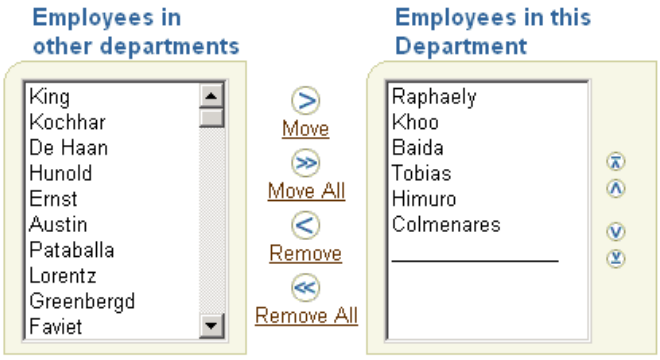
Suggestion: You can solve this by changing the Data Collection of the highest tree group. The Data Collection you specify should contain every possible selectable tree node of this View Object, including the drill-down nodes. If you cannot implement this using a child usage of the parent group, then there is a workaround: temporarily change the layout style of the parent group to tree, then pick any Data Collection 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).

5.7. Creating Shuttle Layouts

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.

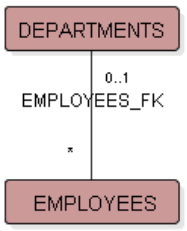
Examples of the use of a shuttle:

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 an **intersection-shuttle**)



5.7.1. 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.

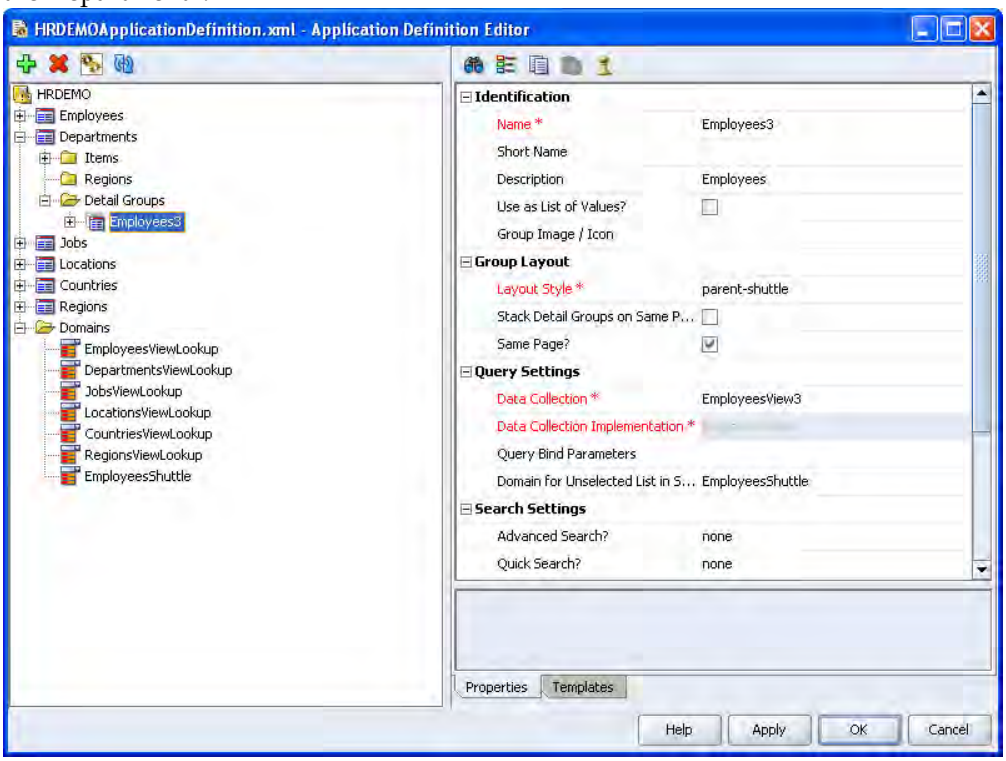
In this example we will create a parent shuttle to assign employees to departments.

Steps to create a parent shuttle:

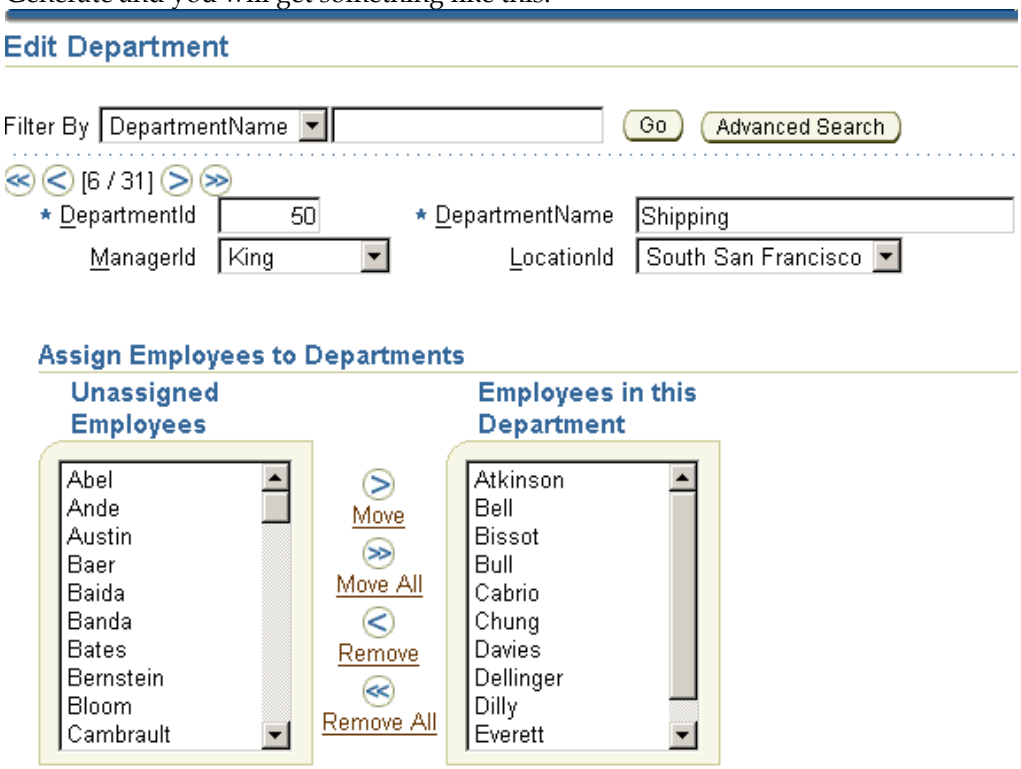
1. Go to the Application Module and add another (top-level) usage of the EmployeesView. Call the usage “EmployeesShuttle”.

2. Create a new Dynamic Domain in the Application Definition Editor. Call it EmployeesShuttle and set its **Data Collection** to EmployeesShuttle. The **Value Attribute** should be EmployeeId and the **Meaning Attribute** LastName

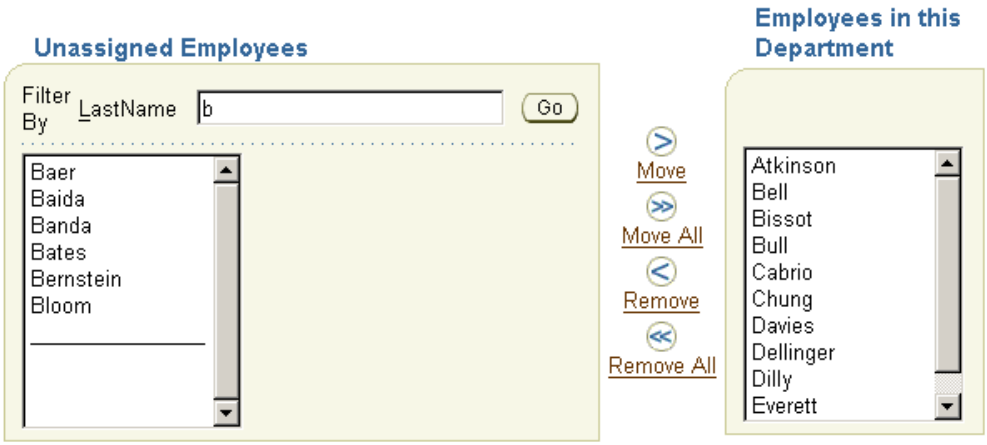
3. Make the following changes to the Employees detail group of the Departments base group. The **Layout Style** of Employees should be parent-shuttle. Set the **Domain for Unselected List in Shuttle** to "EmployeesShuttle". Set the **Tabname** to "Unassigned Employees". Check the property **Same Page?**. Finally set **Display Title (plural)** to "Assign Employees to Departments" and **Display Title (singular)** to "Employees in the Department".



4. Generate and you will get something like this:



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

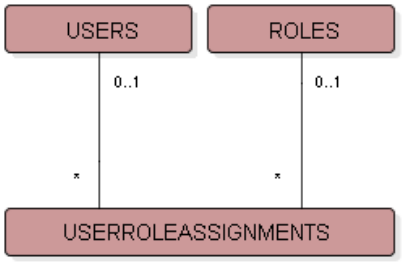


Attention: When deattaching a record (moving from right to left), the employee has no relation with any department anymore. In database terms, the department_id column is set to null. This means it is best to use a parent-shuttle with an optional foreign key.

5.7.2. Creating Intersection Shuttles

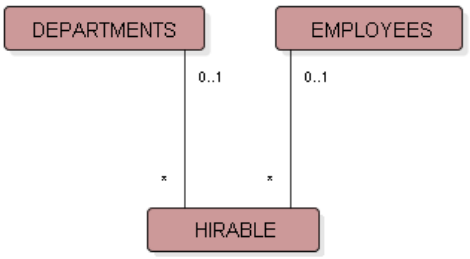
Use an intersection shuttle when you want to maintain an intersection between two ViewObjects. An intersection typically exists when there is a m:n relation between two View Objects. Examples:

- 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 scratch. 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 section 3.2.4 -

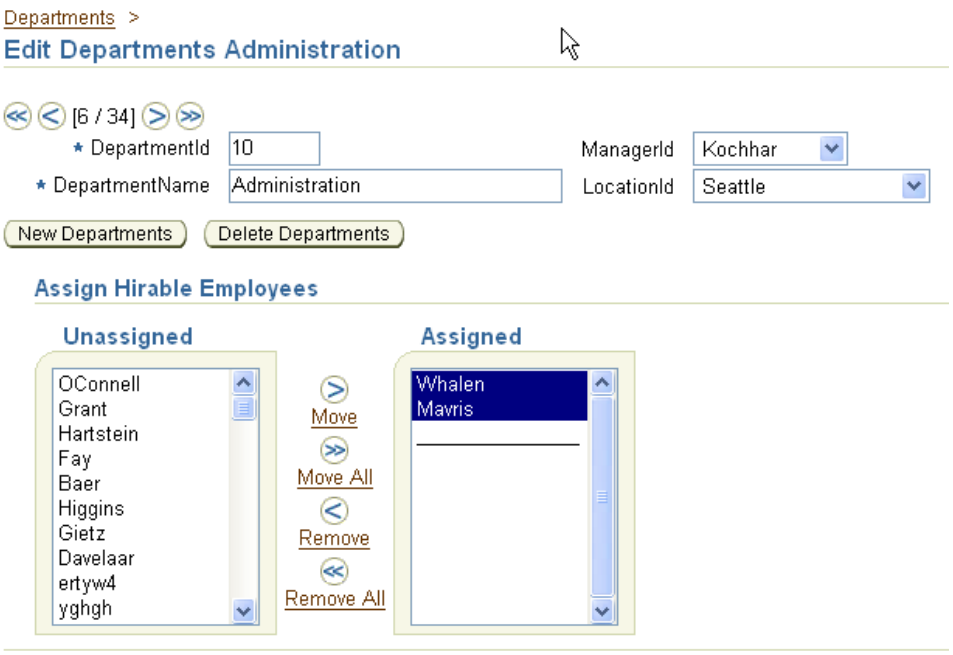
Generating Primary Key Values. Alternatively, you can create an intersection table with a composite primary key consisting of the two foreign key columns.

2. Go to the Application Module and add another (top-level) usage of the EmployeesView. Call the usage "EmployeesShuttle".
3. Create a new Dynamic Domain in the Application Definition Editor. Call it EmployeesShuttle and set its **Data Collection** to EmployeesShuttle. The **Value Attribute** should be EmployeeId and the **Meaning Attribute** LastName
4. Make the following changes to the Hirable detail group of the Departments base group. The **Layout Style** of Hirable should be intersection-shuttle. Check the checkbox **Same Page?**. Set the **Domain for Unselected List in Shuttle** to "EmployeesShuttle". Set the **Tabname** to "Unassigned". Set **Display Title (plural)** to "Assign Hirable Employees" and **Display Title (singular)** to "Assigned".
5. When you now run the JAG, you will get the following error:

   JAG-00126 [Departments / Hirable2] View Object Usage EmployeesShuttle should have a nested View Object Usage based on HirableView.

   The reason you get this error is this: at runtime, JHeadstart needs to know which foreign key attribute(s) in the HirableView map to the primary key attribute(s) of EmployeesView. This information is required to correctly insert a row in the HIRABLE intersection table. JHeadstart uses the ViewLink on which the nested HirableView usage is based to lookup this attribute mapping. So, open te Application Module Editor, and add the HirableView as a nested ViewObject usage under the EmployeesShuttle usage:

6. Generate again. The error should have gone now. Run the application and your page will look like this:



5.7.3. Understanding How JHeadstart Runtime Implements Shuttles

A generated parent shuttle looks like this in the ADF Faces page:

```
<af:selectManyShuttle
    leadingHeader="Unassigned" size="10"
    trailingHeader="Assigned"
    valueChangeListener="#{SubordinatesShuttle.processValueChange}"
    value="#{SubordinatesShuttle.selectedKeys}">
  <af:forEach var="rowbinding"
       items="#{bindings.EmployeesViewLookup.rangeSet}">
    <af:selectItem label="#{rowbinding.LastName}"
       value="#{rowbinding.row.key}"/>
  </af:forEach>
</af:selectManyShuttle>
```

This shuttle element references the SubordinatesShuttle managed bean, which is defined as follows in the group beans faces-config:

```
<managed-bean>
  <managed-bean-name>SubordinatesShuttle</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.bean.ParentShuttleBean
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>selectedRangeBinding</property-name>
    <value>#{bindings.SubordinatesTable}</value>
  </managed-property>
  <managed-property>
    <property-name>jhsPageLifecycle</property-name>
    <value>#{jhsPageLifecycle}</value>
  </managed-property>
  <managed-property>
    <property-name>processShuttleMethodBinding</property-name>
    <value>#{bindings.processSubordinatesShuttle}</value>
  </managed-property>
  <managed-property>
    <property-name>parentChildRefAttrs</property-name>
    <map-entries>
      <map-entry>
        <key>EmployeeId</key>
        <value>ManagerId</value>
      </map-entry>
    </map-entries>
  </managed-property>
</managed-bean>
```

When the user has shuttled entries between the two lists and then submits the page, for example by pressing the Save button, the valueChangeListener method `processValueChange` fires during the Process Validations phase. This method registers the `ParentShuttleBean` instance as a "Model Updater" in `JhsPageLifecycle`. Just before the Model validation phase, `JhsPageLifecycle` calls the `doModelUpdate()` method on the registered "Model Updaters". In method `ParentShuttleBean.doModelUpdate()` the action binding to call the process shuttle

method on `JhsApplicationModuleImpl` (see below) is executed. This action binding is passed in through managed property `processShuttleMethodBinding`.

After calling the action binding the iterator binding of the selected row is requeried, so the newly shuttled rows will be displayed in the correct list after Commit. Note that at this stage the actual database commit has not taken place yet, since this does not happen until Invoke Application phase. However, JHeadstart leverages the ADF BC feature that merges the query result with "open middle tier" changes.

Reference: See the Javadoc or source of `ParentShuttleBean` and `IntersectionShuttleBean`.

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.

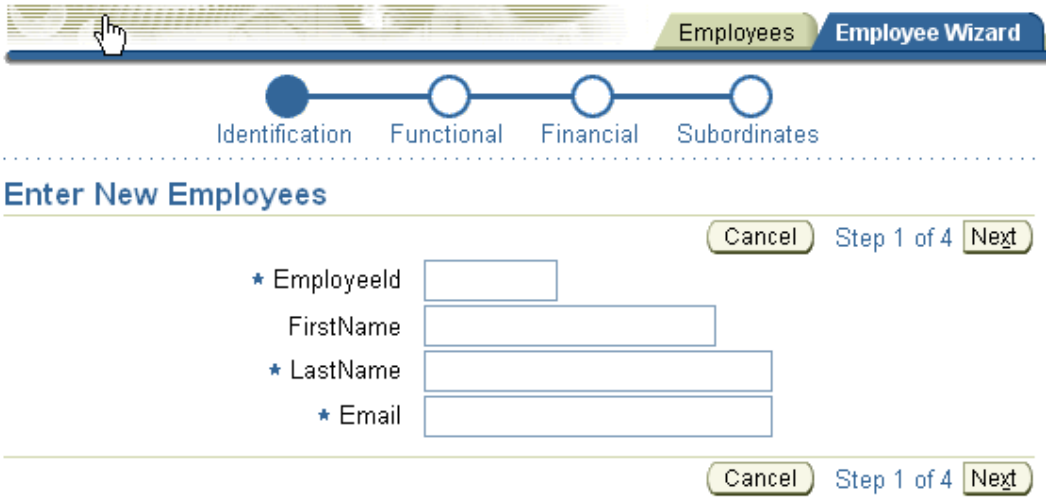
These methods are exported in the Client Interface of the Application Module, which makes them available as Data Control operations. For such an operation an Action Binding can then be created in the UI model of the shuttle page (which is of course what the JHeadstart Application Generator does).

The names of relevant ViewObject 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 or source of `JhsApplicationModule` and `JhsApplicationModuleImpl`, in particular the methods `processParentShuttle()` and `processIntersectionShuttle()`.

5.8. Creating Wizard Layouts

A wizard layout can be used enter a new row in multiple steps. JHeadstart can generate item regions and detail groups to separate steps (pages) of the wizard.



To generate this, we must first ensure that only inserts can be done in the main wizard group.

1. Perform the steps described in section 8.1.3 - Build insert only screens.

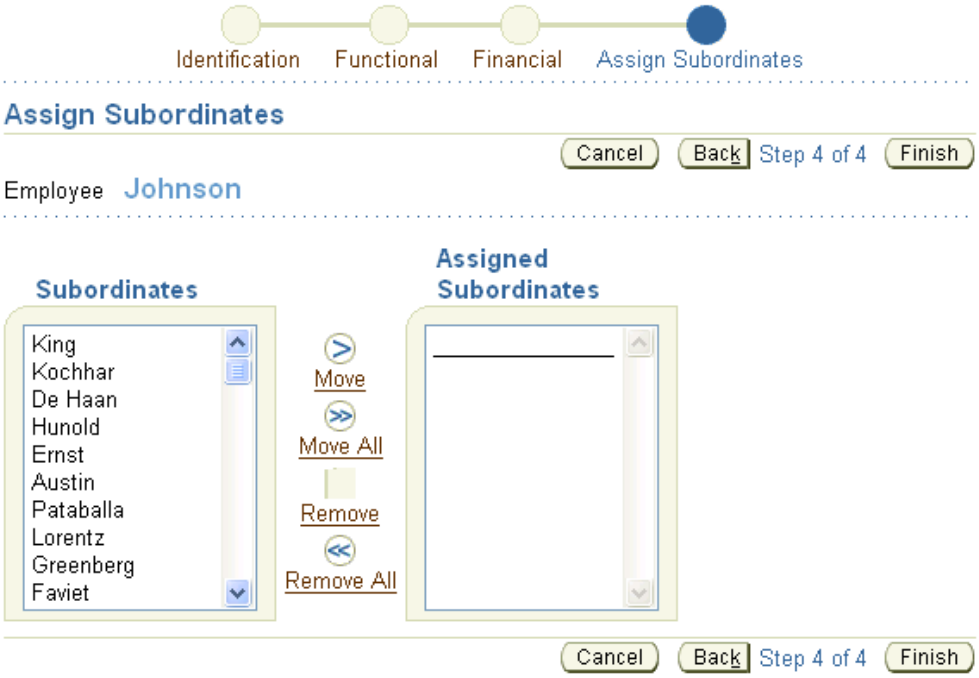
2. In the top-level group's properties, check **Wizard Style Layout?**

3. Create **Item Regions** for the top-level group's items (one region for each wizard step), give the Item Regions a suitable **Title**, and move the items to the regions (see Section 5.1.2 Using Regions).

4. In the **Regions** folder of the top-level group, set the property **Layout Style** to separatePages. This setting allows you spread the items of one group over multiple pages: JHeadstart generates a separate page for each item group within the **Regions** container.

5. Check that detail groups of the top-level group have **Same Page?** unchecked, to ensure that they are generated to separate wizard steps.

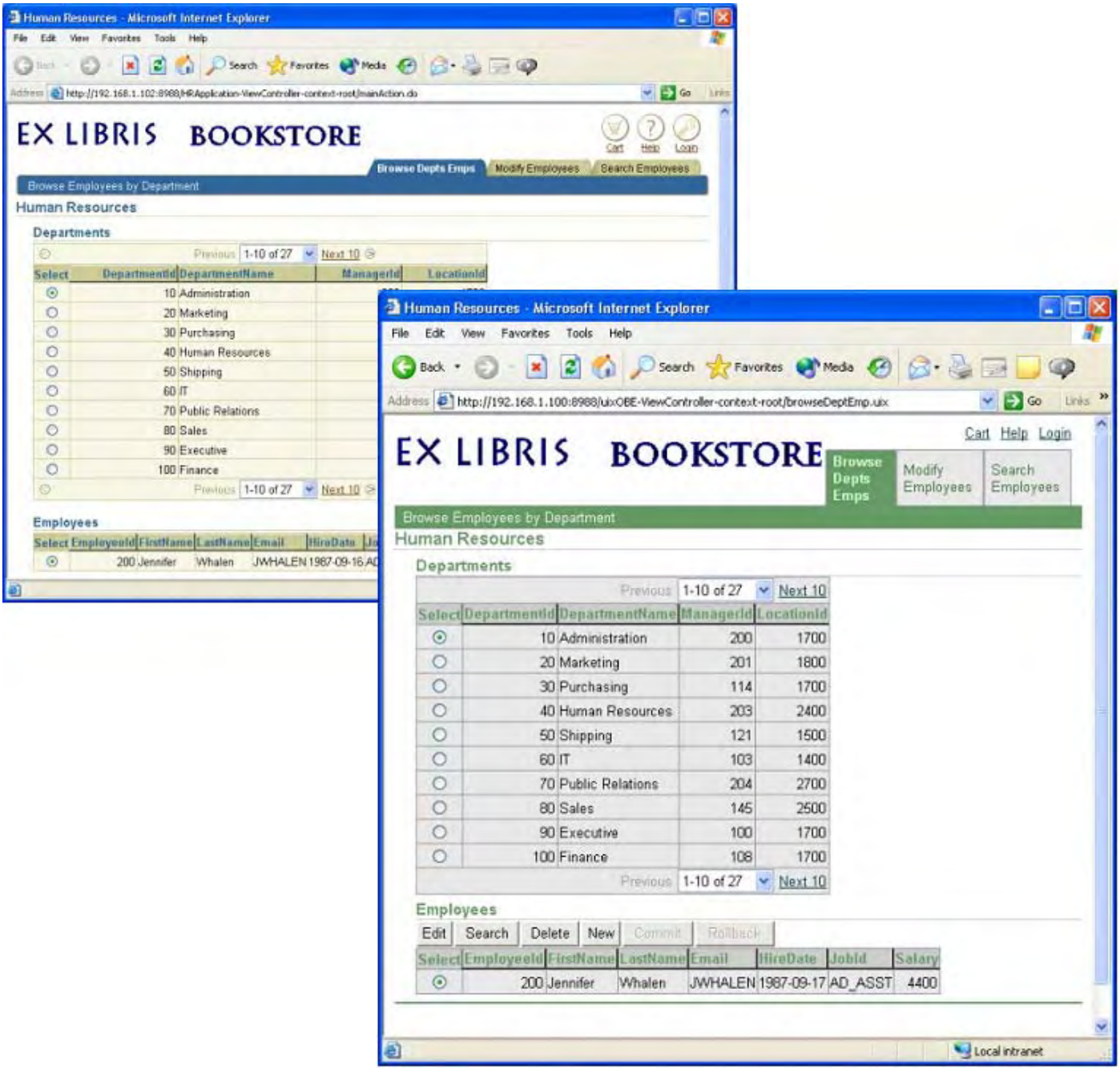
On the last wizard page a Finish button is generated, which will save all changes that were made during the earlier wizard steps.



Attention: Out-of-the-box the wizard style only works for creating new rows. The wizard style generation can also be used for updating existing rows. If you use table-form layout or advancedSearch=separatePage however, you will have to customize the Next button to perform a current row selection or a search, respectively.

5.9. Changing the Overall Page Look and Feel

The overall look and feel of your application, like colors, fonts, margins, icons, button shapes, menu bar, and so forth, is determined by a so-called ADF Faces skin. ADF Faces comes with two default skins "oracle" and "mininal", however you can create your own custom skin to give your application a specific corporate branding.



The advantages of skinning are two-fold:

1. The desired Look and Feel is defined only once in a skin and used throughout one or more applications.
2. You can apply a (new) skin on existing ADF Faces applications, the applications themselves do not need to change.

With a custom skin, you can customize layout characteristics that are typically defined through cascading style sheets: fonts, colors, icons, images, margins and so on. To figure out how you can change the appearance of the various ADF Faces user interface components, this Skin Selectors document is really useful.

Web Reference: Oracle Application Developer Framework Developer's Guide for Forms/4GL Developers Release 10.1.3, section 22.3 "Using Skins to Change the Look and Feel".
http://download.oracle.com/docs/html/B25947_01/web_laf003.htm#CACJAGIG

Web Reference: ADF Faces Skinning Selectors.
http://otn.oracle.com/products/jdev/htdocs/partners/addins/exchange/jsf/doc/skin-selectors.html

Web Reference: ADF Faces Skin best practices.
http://emarcoux.blogspot.com/2007/03/adf-faces-skin-best-practices.html

While skinning is a powerful feature, it only determines the general look and feel of your application. You will need to change some other files to change the application logos/images (see section 5.9.1), and if you want other look and feel characteristics to be generated differently, you can customize the JHeadstart Generator Templates (see section 5.9.2 and 5.9.3).

5.9.1. Customizing the Application Logos

By default, JHeadstart generates pages that show the Oracle and JHeadstart Demo logos. One of the first things you want to customize is probably the replacement of these images by those of your own organization.

Both branding and brandingAppContextual are named facets of the ADF Faces PanelPage component that is used in JHeadstart-generated pages.

The JHeadstart logo's are referenced in the ADF Faces Region files common\regions\branding.jspx and common\regions\brandingAppContextual.jspx. These files were created by the JHeadstart file generator, because the content changes if you use dynamic menus (for more information about dynamic menus, see Chapter 9 "Creating Menu Structures").

Once you have decided if you are going to use dynamic menus or not, you can turn off the file generation for branding.jspx and brandingAppContextual.jspx, so that you can customize them without losing your changes.

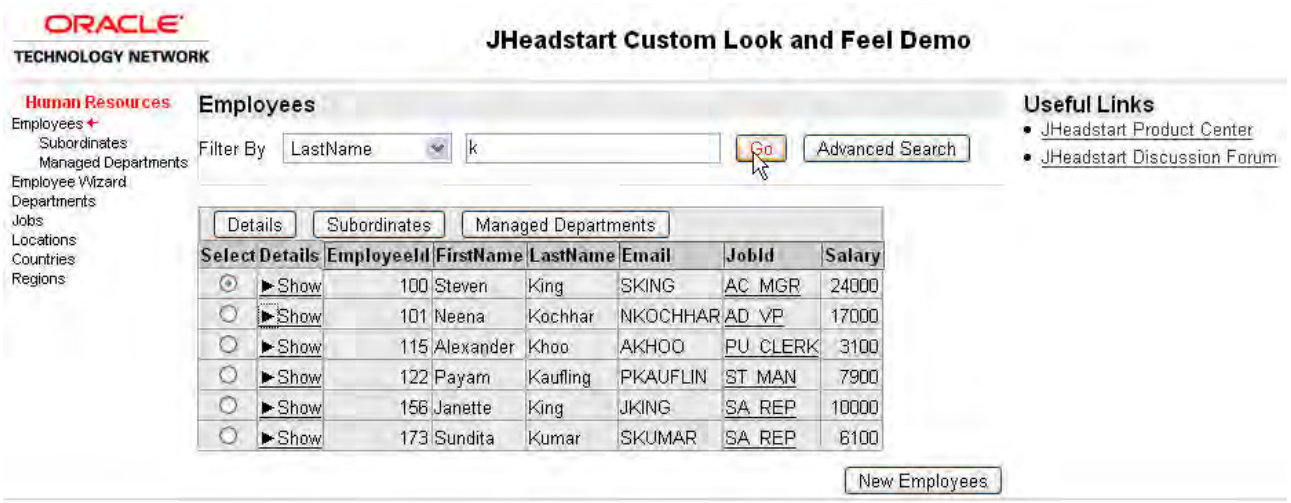
Reference: For instructions how to turn off the generation of these files, see Chapter 4 "Using JHeadstart", section 4.7.6 "The File Generator Template".

After turning off the generation of these files, customize them as follows:

- Open branding.jspx and brandingAppContextual.jspx in ViewController\public_html\common\regions.
- Copy your own logo image(s) to ViewController\images (for example, use SRBranding.gif from the Oracle SR Demo).
- Change brandingAppContextual.jspx and branding.jspx to refer to your own image instead of the JHeadstart image, or remove the image reference if you don't want an image in that facet.

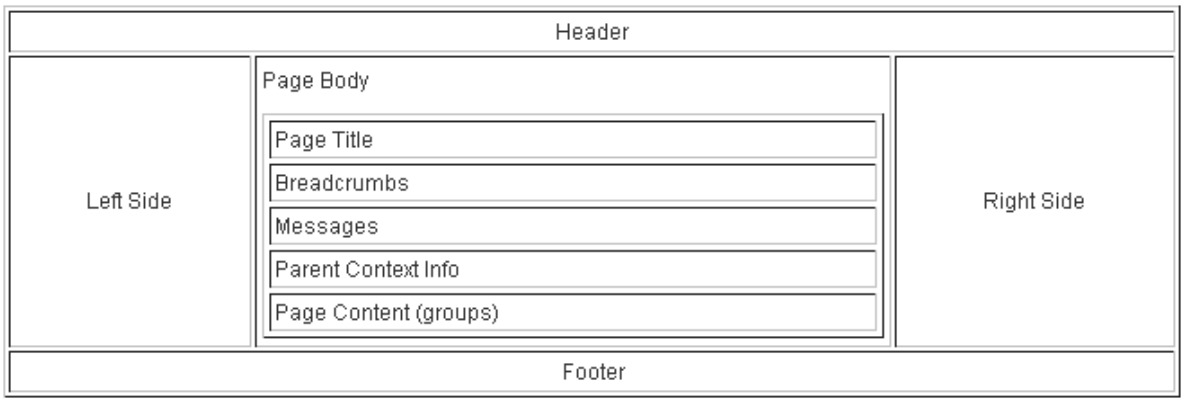
5.9.2. Rearranging the Overall Page Layout Using Generator Templates

While ADF Faces skinning is a nice feature, it does not allow you to perform more "radical" layout customizations that include rearranging the overall page layout. For example, in many web sites, the menu is shown in a nested structure at the left side of the page, as shown in the example layout below.



Customizations like this can easily be achieved by creating a set of custom Velocity templates and configure the JHeadstart Application Generator to use this custom set of templates. For more information, see Section 4.7 Using Generator Templates.

The basic idea is that instead of the ADF Faces PanelPage component, you create your own page structure, for example like this:



The content in the header, footer, leftSide, and rightSide areas is defined in ADF Faces regions by the same name, to prevent the actual content of these areas from being duplicated in each and every page.

Reference: You can download this example. See the JHeadstart Blog post 'Generating a Custom Look and Feel using JHeadstart' at http://blogs.oracle.com/jheadstart/2006/12/22 - a122.

5.9.3. Creating Custom ADF Faces Regions and using them in Generator Templates

In ADF Faces 10.1.3, Regions are snippets of ADF Faces source code that are stored in separate files, and can be reused in several ADF Faces pages. JHeadstart uses these ADF Faces Regions for the application branding, the menu sections, and for example the form browse buttons (arrow buttons to navigate to the previous or next record in form layout).

In general, you have to perform the following steps to customize an ADF Faces Region, if you want to make a copy of the original region:

1. Create custom ADF Faces Region file
2. Declare the new Region in region-metadata.xml
3. Create customized version of Generator Template file that references the region, in which you refer to the new Region
4. Refer to customized Generator Template in Application Definition

These steps are detailed below for customizing the application logo(s). As explained in section 5.9.1 Customizing the Application Logos, it is not necessary to use this technique for the logos because JHeadstart does not overwrite the relevant region files, but it is useful to use this example for explaining the steps.

5.9.3.1. Creating a Custom Region

The JHeadstart logo's are referenced in the ADF Faces Region files common\regions\branding.jspx and common\regions\brandingAppContextual.jspx. As the golden rule is that you should not change files that were created by JHeadstart, the way to do this is to copy these branding region files and make customized versions of them.

- Copy branding.jspx and brandingAppContextual.jspx in ViewController\public_html\common\regions to for example customBranding.jspx and customBrandingAppContextual.jspx. If you want to replace the 2 images by just 1, you only need to copy brandingAppContextual.jspx.
- Copy your own logo image(s) to ViewController\images (for example, use SRBranding.gif from the Oracle SR Demo).
- Change customBrandingAppContextual.jspx (and possibly customBranding.jspx too) to refer to your own image instead of the JHeadstart image.

5.9.3.2. Declaring the New Region

To declare these new jspx files as ADF Faces Regions, you must modify WEB-INF/region-metadata.xml.

- Open WEB-INF/region-metadata.xml, find the component `oracle.jheadstart.region.brandingAppContextual` (and possibly `oracle.jheadstart.region.branding` too), and copy the complete component definition. Change the name of the component to, for example, `com.mycompany.myapp.view.region.brandingAppContextual`, and change the reference to the jspx file of the region.

5.9.3.3. Customizing the Data Page Template

Now that you have customized the branding region(s), you must ensure that the new regions are used in every generated page. You can do this by creating a customized version of the data page generator template (see also section 4.7 Using Generator Templates).

- In your ViewController project, find the Resource default/page/dataPage.vm.
- Open it, and Save As custom/page/dataPage.vm (for example).
- In the copied dataPage template, optionally remove the `<f:facet name="branding">` (if you only want the brandingAppContextual). Change the `<f:facet name="brandingAppContextual">` to refer to the new region (set the regionType to the name of the new component you defined in region-metadata.xml, for example `com.mycompany.myapp.view.region.brandingAppContextual`).

5.9.3.4. Using the Customized Template in the Application Definition

The last step is to specify this custom dataPage.vm in the Application Definition (see also section 4.7 Using Generator Templates).

- Open the Application Definition editor.
- On Service level, go to the Templates tab.
- For DATA_PAGE, enter custom/page/dataPage.vm

Now if you generate the application again, each page will have a header like this:

This page is intentionally left blank.

CHAPTER

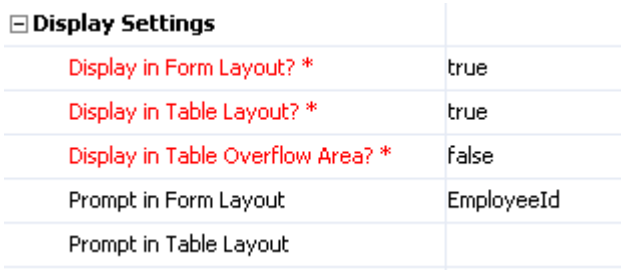
6

Generating User Interface Widgets

This chapter 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.

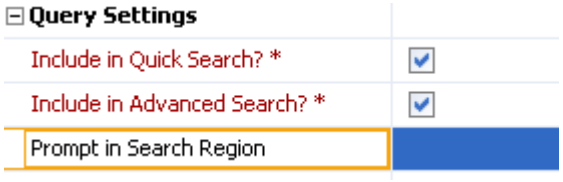
6.1. Specifying the Prompt

If you don't include a **Prompt in Form Layout**, the generated page will not have a label for that field. If you want to display a prompt, you specify this using the **Prompt in Form/Table Layout** property for form/table pages.



If you don't include a **Prompt in Table Layout**, it will default to the **Prompt in Form Layout**.

A separate property **Prompt in Search Region** allows you to override the **Prompt in Form Layout** in search areas:

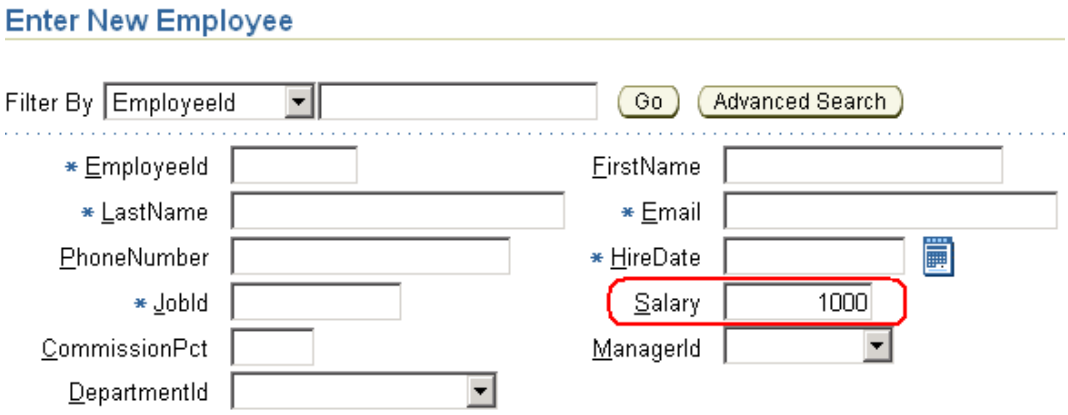


6.2. Default Display Value

In the Application Definition Editor you can set the **Default Display Value** of an item. 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 Application Definition Editor.
2. Generate/run your application and create a new record. The default display value is shown now in the column.



6.2.1. Using EL expressions

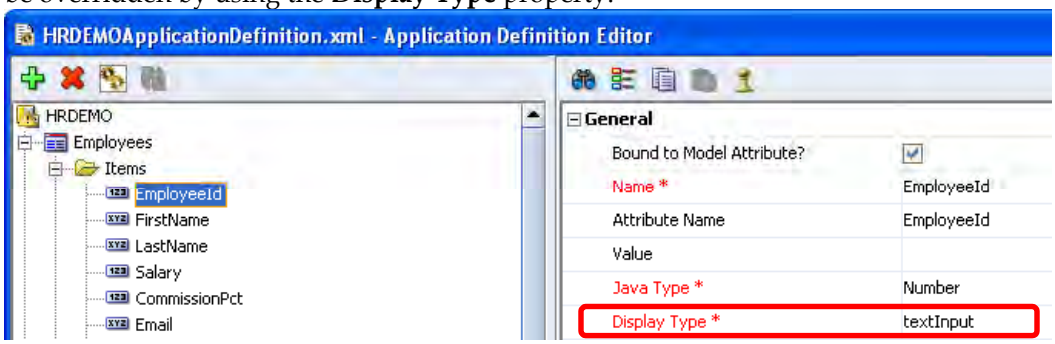
In addition to literal values, you can enter Expression Language in the **Default Display Value.**

Imagine that for new employees the salary must be calculated based on job and so on. Assume the result of the calculation 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.

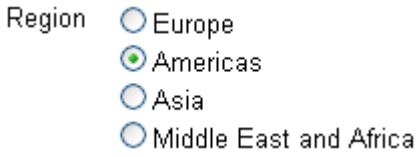
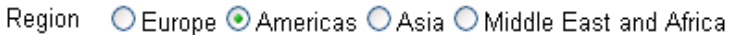
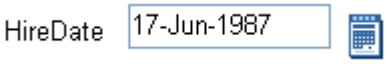
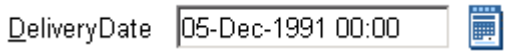
6.3. Display Type

Default display types are set during the creation of the Application Definition. These can be overridden by using the **Display Type** property.



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

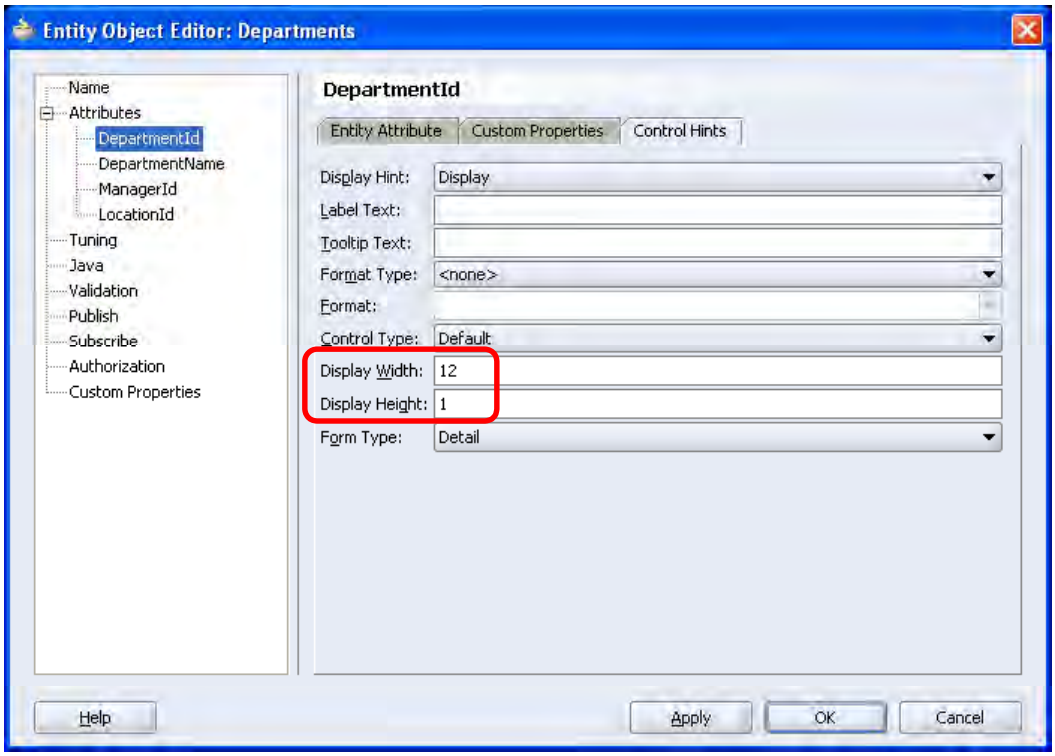
textInput	* City Roma
dropDownList	Region Europe Americas Asia Middle East and Africa
lov	Department Executive
checkbox	Lease Car?
graph	16K 14K 12K 10K 8K 6K 4K 2K 0K Salary Russell Partners Errazuriz Cambrault Zlotkeys
list	Commission % No Normal High

radio-vertical	
radio-horizontal	
editor	
dateField	
datetimeField	
secret	
fileUpload	
fileDownload	
image	
hidden	
flexRegion	See Chapter 12 "Runtime Page Customizations"
oraFormsFaces	See section 6.15 “Embedding Oracle Forms in JSF pages”

6.4. Generating a Text Item

6.4.1. Define Item Display Width and Height

The item display width and height can be set in two ways. The first option is dynamical. EL expressions are used to get the width and height from the underlying ADF Business Components. When these properties are not set on the Entity Object, the View Object is checked for these properties.



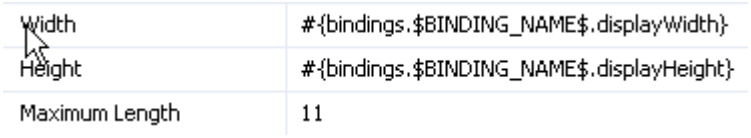
By default, an item is displayed with height = 1 (line) and width = the data length of the underlying table column. When the length of the table column is unknown or larger than the value of the service level property **Default Display Width**, the value of this property is used.

Display Settings	
Display in Form Layout? *	true
Display in Table Layout? *	true
Display in Table Overflow Area? *	false
Prompt in Form Layout	DepartmentId
Prompt in Table Layout	
Width	#{bindings.DepartmentsDepartmentId.displayWidth}
Height	#{bindings.DepartmentsDepartmentId.displayHeight}

The second option is the static option. The **Width** and **Height** properties can be used to hardcode the values for width and height. So instead of using the EL expressions, use static numbers.

6.4.2. 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** property of the item. The value should be the number of characters you require, or an EL expression returning such a number.



6.5. 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 Static Domain or a List Validator.
- The list of values is dynamic. In this case you must base the dropdown list on a Dynamic Domain.

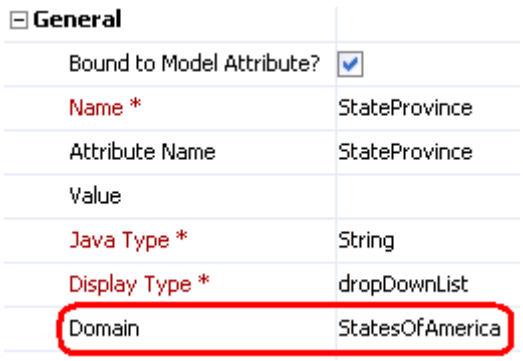
6.5.1. Static dropdown list based on a Static Domain

When using this option, you have to add your domain with its values to the Application Definition Editor.

You can create a static domain by selecting the Domains node in the Application Definition Editor. After pressing the green plus (+) symbol select the static domain. A domain with the name newStaticDomain is created. This name can easily be changed in something more descriptive.

The last step is adding of values (and their meanings) to the new domain. An undefined value is already provided with the newStaticDomain. After changing this value new values can be added by selecting the new static domain and pressing the green plus (+) symbol. See also the section Domains.

To use the newly created domain set the **Display Type** of an item to dropDownList, radio-vertical or radio-horizontal and fill its **Domain** property with the name of the new Domain.



6.5.2. Translation of static domains

The meaning of the domains in the Application Definition Editor is only in one 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.

Reference: Chapter 11 “Internationalization and Messaging”, section “National Language Support in JHeadstart”

6.5.3. Dynamic dropdown list based on a Dynamic Domain

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

Steps to generate a dropdown list based on a Dynamic Domain:

1. Create a Dynamic Domain based on the View Object. Select the View Object Usage (in the data model of the Application Module) you want, by setting the **Data Collection** property for the Dynamic Domain.
2. Set the **Value Attribute** of the Dynamic Domain to the attribute you want to store in the item (which uses the domain).
3. Set the **Meaning Attribute** to the attribute you want to show in the dropdown list.

General	
Domain Name *	EmployeesViewLookup
Domain Type *	Dynamic
Query Settings	
Data Collection	EmployeesViewLookup
Data Collection Implement...	EmployeesView
Dynamic Data Collection E...	
Query Bind Parameters	
Display Settings	
Value Attribute	EmployeeId
Meaning Attribute	LastName

4. Set **Display Type** of the item that uses the domain to 'dropDownList' (or 'radio-vertical/radio-horizontal').
5. Set the **Domain** property to the Dynamic Domain.

General	
Bound to Model Attribute?	☑
Name *	ManagerId
Attribute Name	ManagerId
Value	
Java Type *	Number
Display Type *	dropDownList
Domain	EmployeesViewLookup

6.6. Generating a Radio Group

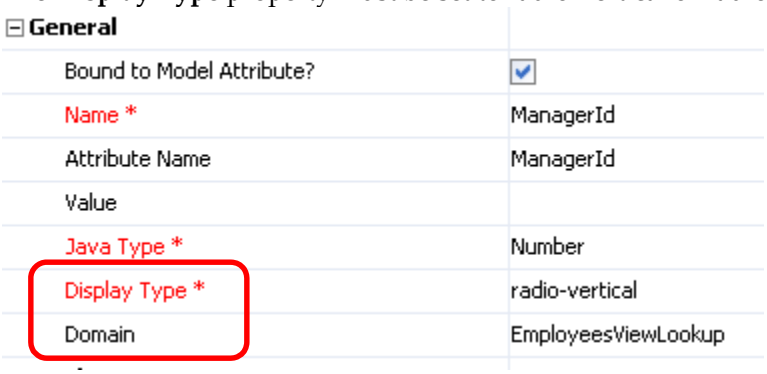
Use a radio group when the list of values the user can choose from is small. You have to distinguish between two cases:

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

6.6.1. Static radio group based on a domain

When using this option, you have to create a Static Domain as described in Static dropdown list based on a Static Domain.

The **Display Type** property must be set to radio-vertical or radio-horizontal.



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

6.6.2. Translation of static domains

As you see, the meaning of the domains in the Application Definition Editor is only in one 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.

6.6.3. Dynamic radio group based on a Dynamic Domain

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

The steps to create a dynamic radio group are almost identical to the steps for creating a dynamic dropdown list, see Dynamic dropdown list based on a Dynamic Domain. The **Meaning Attribute** is used to get labels for the radio buttons. Finally the **Display Type** should be radio-vertical or –horizontal.

6.7. Generating a List of Values (LOV)

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 dropdown list or you want to provide search functionality on the lookup.

An LOV is actually just a group as all other groups only it has the **Use as List of Values?** checked and the **Layout Style** set to table. This kind of group is called a *LOV group*.

Furthermore an LOV is always attached to an item. This item gets a little lantern next to it in the web pages. This item is called the *LOV item*.

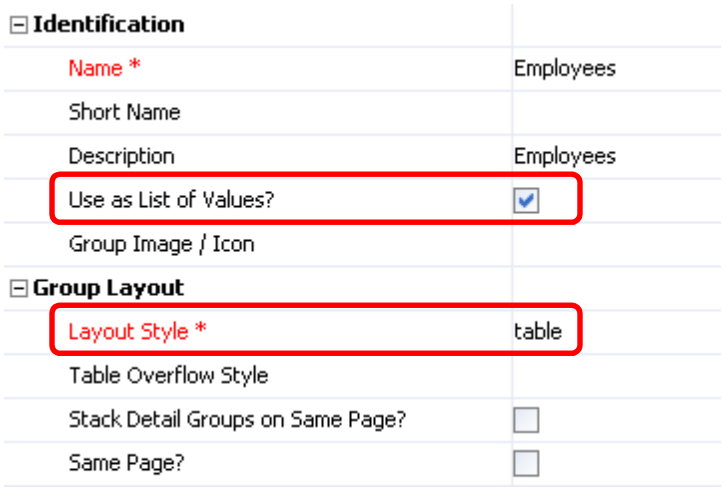
Normally this LOV item (**Target Item**) is filled with a value from the LOV group (**Source Item**). The mapping of Source Item to Target Item is what we call a *return value*. An LOV can have several return values.

Steps to create a list of values:

1. Create a (reusable) LOV group
2. Link the LOV group to an item

6.7.1. Creating a (reusable) LOV group

1. Create (or reuse) a base group that will contain the rows of the LOV (which will be used as LOV group). The creation of a base group is explained in Creating objects.
2. Set the **Layout Style** (of the LOV group) to 'table'.
3. .Check the **Use as List of Values?** property.



4. Specify at least one type of search for the LOV group: quick search or advanced search.
5. Set the **Display Type** of the LOV item (which will have a LOV attached) to lov.

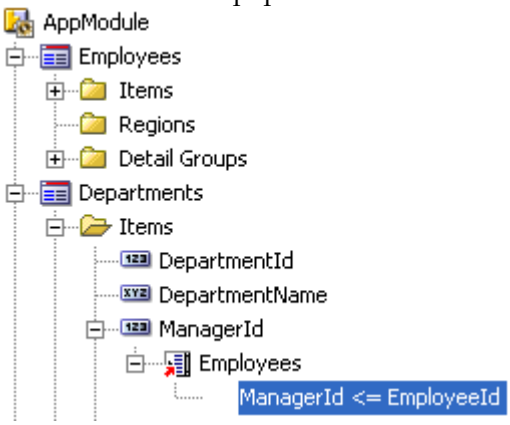
Attention: If the Data Collection of your LOV Group is based on a Read-Only ViewObject that is not based on an Entity Object, then do the following:

- Make sure at least one attribute in the ViewObject is marked as Key Attribute
- Override method `create()` in your ViewObjectImpl class, and in this method call `setManagerRowsByKey(true);`

If you do not perform these steps, the LOV will not copy back any values!

6.7.2. Linking a (reusable) LOV group to an item

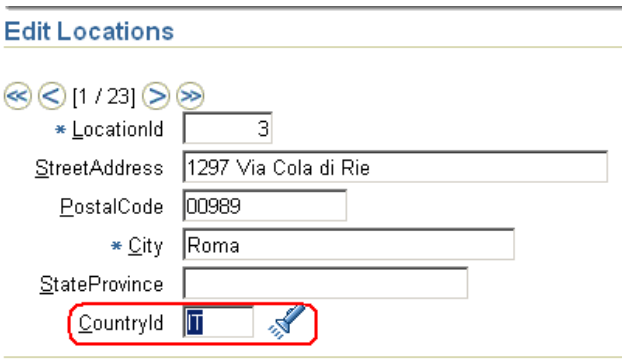
1. Select the LOV item (and press the green plus (+) symbol.
2. Set the **LOV Group Name** property, and in the first return item, set the **Source Item** property. The Source Item should be set to the item from the LOV group that will be used to populate the LOV item.



3. Generate and you will get something like this. Here the Employees group is used as LOV group. ManagerId is the LOV item and target item. The source item is EmployeeId (from the LOV group).

6.7.3. Defining an LOV on a display item

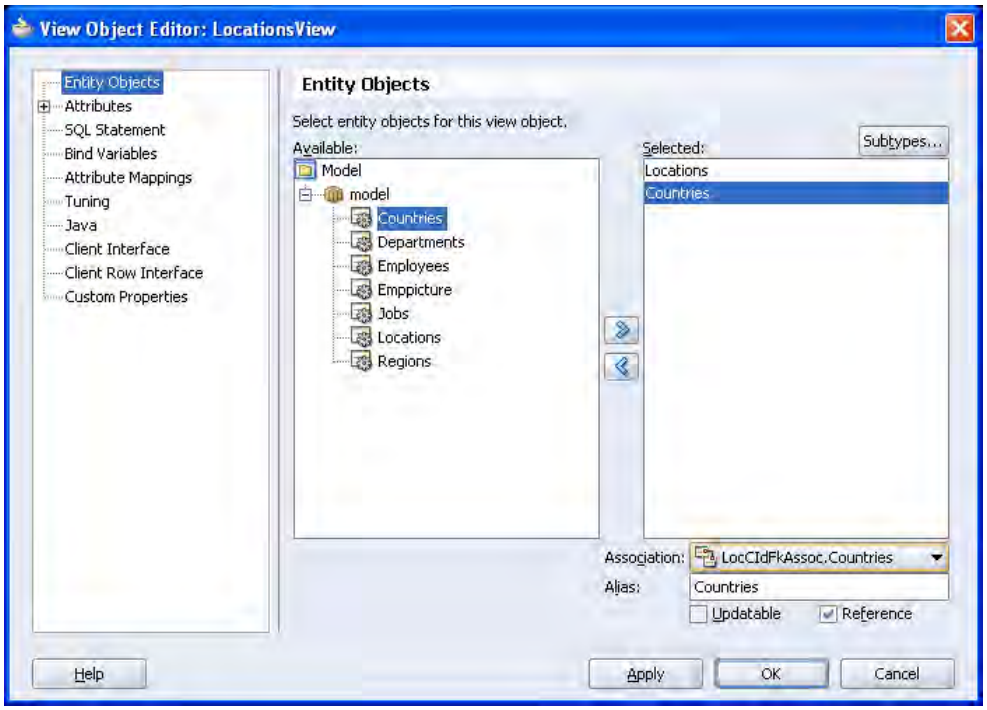
There are situations where you will need to define lookup attributes in the base view object. Take a look at the screen below. The CountryId column is part of the Locations View Object. The CountryId is a foreign key referencing the Countries View Object. However, in many (most) cases, you do not want 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 Definition. So the CountryName attribute of the Countries View must be added to the LocationsView.

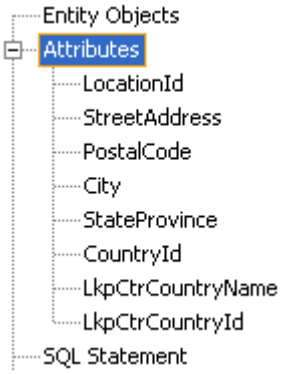
Perform the following steps to accomplish this:

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 (dropdown list just below the Selected Entities box).
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. Uncheck the Key Attribute checkbox of all "Lookup" key attributes that are automatically added as described in the previous step (see also section 3.3.3.1. Unchecking Reference Key Attributes for Updateable View Objects). **If you forget this step, the LOV lookup values will not be visible when returning from the page.**
7. 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.



8. Test the View Object with the ADF BC Tester (see section 3.3.9 "Testing the Model"). Check whether the LkpCtrCountryName attribute changes when you change the CountryId attribute.

6.7.3.1. What to do when ADF BC Tester does not update the lookup item

In this example, the LkpCtrCountryName will most likely be updated correctly in the ADF BC tester. However, in your application this might not work due to one of the following causes:

- **Cause**: You did not define the View Object's lookup entity usage on the right Entity Association (end).
  Solution: Correct the View Object Definition.

- **Cause**: There is no underlying entity association, for example because the LOV ViewObject is a read-only ViewObject.
  Solution: In this case, you need to perform some additional steps. These steps are explained below using the same Locations/Countries example as above, although they are not required for this specific example. **So, you only need to do the 4 additional steps below in your own application if this cause applies**.

1. Open the Locations ViewObject and create an additional transient attribute named "LkpCtrCountryNameTransient". Set updateable to "always" and uncheck the queryable checkbox.
2. In the Java Tab check the checkbox to create a LocationsViewRowImpl java class. Click OK to close the ViewObject editor

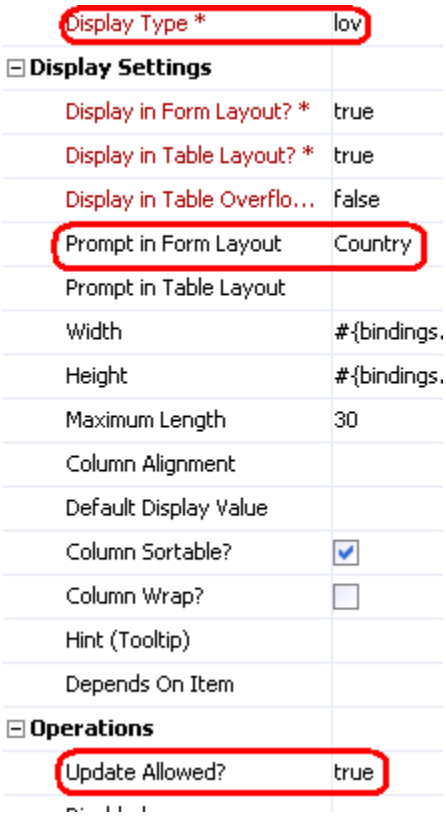
3. Got to the newly created LocationsViewRowImpl class, and modify the getLkpCtrCountryNameTransient method as follows:

```
public String getLkpCtrCountryNameTransient ()
  {
    if (getAttributeInternal(LKPCTRCOUNTRYNAMETRANSIENT)==null)
    {
      return getLkpCtrCountryName();
    }
    return (String) getAttributeInternal(LKPCTRCOUNTRYNAMETRANSIENT);
  }
```

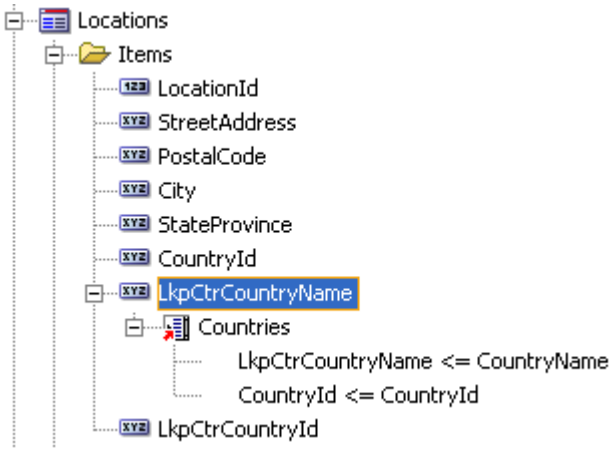
4. Continue with the steps below, but base your List of Values on the LkpCtrCountryNameTransient item rather than on the LkpCtrCountryName item. Hide the LkpCtrCountryName item in your pages by setting both **Display in Form** and **Display in Table** to false for this item.

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

1. On the Countries group check the **Use as List of Values?** property.
2. Select the Locations group and press the Synchronize button (the circular blue arrows). This will add the newly created attributes as items to the group.
3. Set the **Display Type** of LkpCtrCountryName (or LkpCtrCountryNameTransient) to lov.
4. Change the **Prompt in Form Layout** property of LkpCtrCountryName (or LkpCtrCountryNameTransient) to Country.
5. Enable updates on LkpCtrCountryName (or LkpCtrCountryNameTransient) by setting the **Update allowed?** property to true.

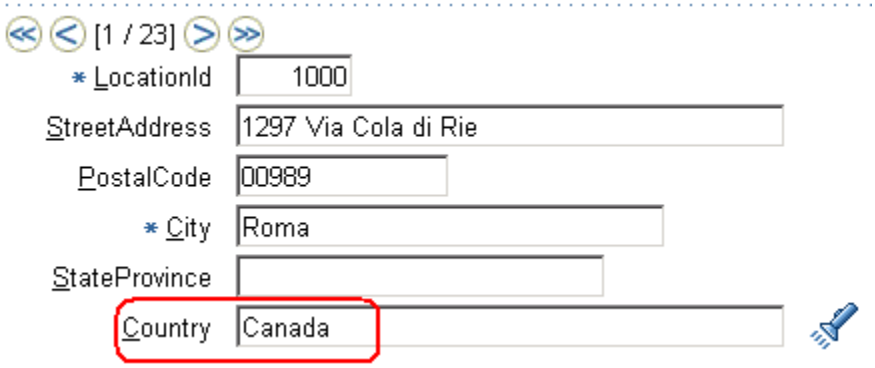


6. Select item LkpCtrCountryName (or LkpCtrCountryNameTransient) and add a LOV as described above in Linking a (reusable) LOV group to an item. Use Countries as LOV group and CountryName as source item.
7. Select the LOV Countries (under LkpCtrCountryName) and add another return value by pressing the green plus (+) symbol.
8. Change undefined <= undefined to CountryId <= CountryId



9. Make CountryId and LkpCtrCountryId invisible by setting the **Display in Form/Table Layout?** properties to false.

10. Generate and you get something like this.



6.7.4. Use LOV for Validation

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:

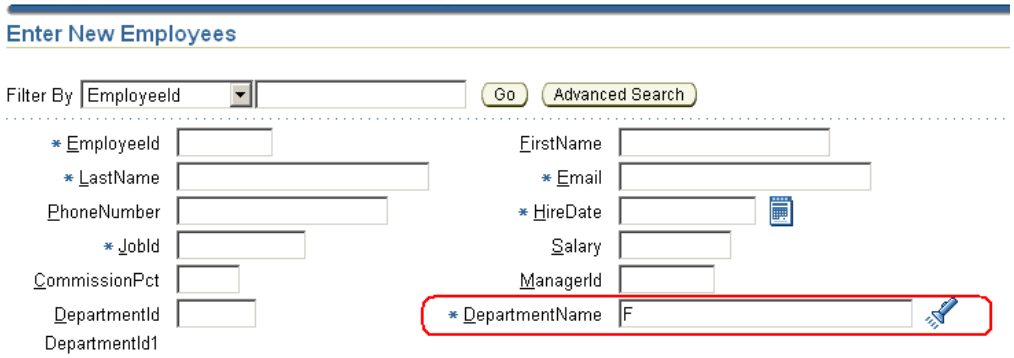
1. The user enters (part of) the lookup item value.
2. The JHeadstart runtime checks how many records in the lookup match the value the user entered.
3. When it is exactly one, the list of values window is not shown, but the JHeadstart runtime finds the matching record and auto-completes the entered 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. The goal is to see the department name in the Employees page.

1. Extend the base View Object you want to manipulate with the descriptor attributes of the lookup View Object. See Defining an LOV on a display item for instructions. In our example, the DepartmentName attribute should be added to the EmployeesView View Object. Use LkpDptDepartmentName and LkpDptDepartmentId as identifiers for the new columns.
2. In the JHeadstart Application Definition, define a group for the base View Object (Employees). Set group properties as you like.
3. Define a LOV group for looking up the department name (see Creating a (reusable) LOV group).
4. Add an LOV on the LkpDptDepartmentName (see Linking a (reusable) LOV group to an item). Set the **LOV Group Name** to Departments and the **Source Item** to DepartmentName.

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

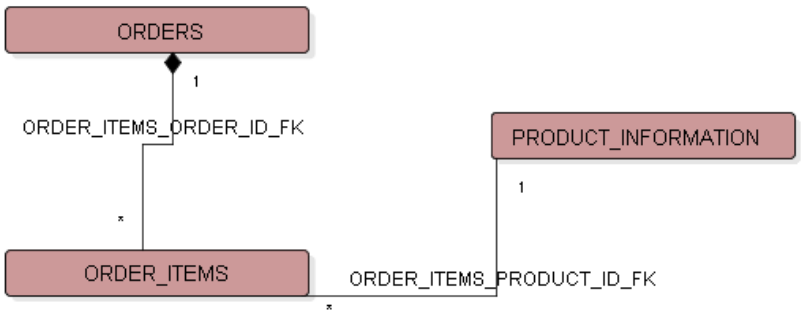


- Navigate to the DepartmentName, enter 'F' in the field and press TAB. Because there is only one department name starting with F, no LOV will be launched and the department name is auto completed.
- Navigate to the DepartmentName, enter 'CO' in the field and press TAB. Because multiple department names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

6.7.5. Selecting multiple values in a List of Values

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.

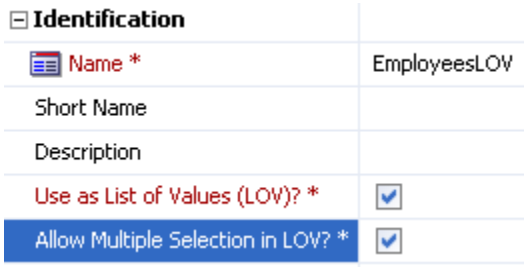
How to generate this:

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 section 3.2.4 - Generating Primary Key Values.

2. Make sure you have groups defined correctly. In this example, ORDERS can have form layout, ORDER ITEMS *must* have table layout and PRODUCT_INFORMATION is an LOV group with table layout.
3. In the base group of the lov (in this example ORDER ITEMS), set **Multi-row Insert Allowed?** and give the **New Rows** property a value greater than zero.

4. In the LOV group, check the boxes for **Use as List Of Values (LOV)** and **Allow Multiple Selection in LOV?**.



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.

Suggestion: An alternative user interface for this situation is an Intersection Shuttle. See section 5.7.2. Creating Intersection Shuttles.

6.7.6. Understanding How JHeadstart Runtime Implements List Of Values

When you specify a ListOfValues element for a group item, JHeadstart generates one or more managed bean definitions for this LovItem. The number of beans depends on whether the item is used in a search region, in a table layout and/or a form layout. Here is an example of a managed bean definition for an LOV Item in a form layout:

```
<managed-bean>
  <managed-bean-name>DepartmentsManagerNameLovItem</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.bean.LovItemBean
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>lovFieldBinding</property-name>
    <value>#{bindings.DepartmentsManagerName}</value>
  </managed-property>
  <managed-property>
    <property-name>lovPage</property-name>
    <value>#{ManagersLovPage}</value>
  </managed-property>
  <managed-property>
    <property-name>lovSearchBean</property-name>
    <value>#{searchManagersLov}</value>
  </managed-property>
  <managed-property>
    <property-name>returnValues</property-name>
    <map-entries>
      <map-entry>
        <key>LastName</key>
        <value>DepartmentsManagerName</value>
      </map-entry>
      <map-entry>
        <key>Email</key>
        <value>DepartmentsManagerEmail</value>
      </map-entry>
      <map-entry>
        <key>EmployeeId</key>
        <value>DepartmentsManagerId</value>
      </map-entry>
    </map-entries>
  </managed-property>
  <managed-property>
    <property-name>copyToModel</property-name>
    <value>true</value>
  </managed-property>
</managed-bean>
```

The returnValues property is used to copy back the correct values to the base page, and/or bindings of the base page.

This bean is referenced by multiple properties in your LOV Item, as shown below:

```
<af:selectInputText
      id="DepartmentsManagerName" label="ManagerName"
      partialTriggers="DepartmentsManagerName"
      required="false"
      columns="#{bindings.DepartmentsManagerName.displayWidth}"
      maximumLength="25"
      value="#{DepartmentsManagerNameLovItem.lovFieldValue}"
      autoSubmit="true" immediate="true"
      valueChangeListener="#{DepartmentsManagerNameLovItem.validateWithLov}"
      binding="#{DepartmentsManagerNameLovItem.lovField}"
      action="dialog:ManagersLov" windowHeight="200"
      returnListener="#{DepartmentsManagerNameLovItem.returnedFromLov}"/>
```

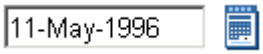
When you have selected a row in the LOV, and the LOV window is closed, the returnListener method `returnedFromLov()` is executed. This method calls method `copyReturnValues()` in the same bean class (LovItemBean), which reads the returnValues property, and copies back the values.

When you have checked the **Use LOV For Validation?** checkbox on the item's List of Values element in the Application Definition Editor, the valueChangeListener and autoSubmit properties are generated, like in the above example. These two properties cause the page to be submitted when the user tabs out the LOV item after changing the value, and JSF calls the valueChangeListener method `validateWithLov()`. This method uses the SearchBean of the LOV page (provided through the `lovSearchBean` managed property) to execute a query against the LOV ViewObject Usage. When the query returns exactly one row, validation is successful and the `copyReturnValues()` method is called to copy back the values. When zero or more than one row is returned by the query, the LOV window is launched.

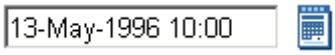
Reference: See the Javadoc or source of `LovItemBean`.

6.8. 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.



6.8.1. 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 the ApplicationResources.properties file (under datepattern and datetimepattern). This file is used at runtime. In case of changes in the Internationalization properties on the service level, these properties can be stored in another locale.

6.9. Generating a Checkbox

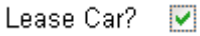
You can generate a checkbox for attributes that have exactly two allowable values: one value is shown as checked, and the other as unchecked. Because the HR sample schema does not have such an attribute, we have added IND_LEASE_CAR to the EMPLOYEES table, with allowable values Y and N.

Steps to generate a checkbox:

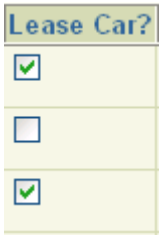
1. Create a static domain with exactly two values (see the section Domains). The first value in the domain is the checked value, the second value in the domain is the unchecked value.

2. In the properties of the item that you want to generate as a checkbox, set **Display Type** to 'checkbox' and **Domain** to the static domain you just created (for example YesNo).
3. Set the **Default Display Value** to one of the values in the Domain.
4. It is customary to change the **Prompt** to something ending in a question mark, for example 'Lease Car?'.

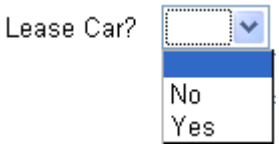
This will generate a field like this in form layouts:



In a table layout it will look like this:



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

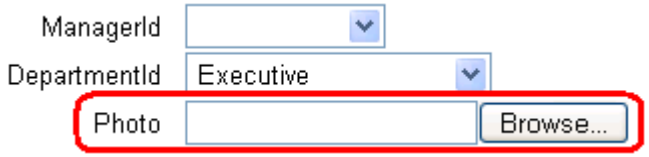
6.10. File Upload, File Download, Showing Image Files, and Playing Audio Files

You can generate File Upload, and depending on the type of file, File Download or Show Image or Play Audio for database columns of types ORDSYS.ORDDOC, ORDSYS.ORDIMAGE and ORDSYS.ORDAUDIO. If you have stored your files in BLOB columns instead, see section Using JHeadstart File Up/Download on BLOB Columns.

Attention: The abovementioned types are object types defined in the Oracle *inter*Media feature of the Oracle database. The ORDSYS.ORDDOC type can store any heterogeneous media data including audio, image, and video data in a database column.
ORDSYS.ORDIMAGE can process and automatically extract properties of images of a variety of popular data formats, and ORDSYS.ORDAUDIO can process audio specific properties.
For more information, see the *inter*Media section of the Oracle Technology Network (http://otn.oracle.com/products/intermedia).

Example of generating a file upload field in the HR sample schema for uploading photos of employees:

1. Make sure you have a table with a column of the correct datatype. This is sufficient:
```
alter table EMPLOYEES add photo ordsys.ordimage;
```
2. Add the new Photo attribute to the ADF Entity Object and ADF View Object for Employees.
3. Add the Photo item to the Employees group of your JHeadstart Application Definition (by synchronizing the group) 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.

3. If instead, you want that field to display the photo, you can change the **Display Type** of the item to *image*.
4. If instead, you want that field to display a hyperlink that downloads the file in a separate window, you can change the Display Type of the item to *fileDownload*.

Attention: The display type 'image' means that ADF Faces renders the file as a download link, image, or audio player, depending on the nature of the individual file.

After changing the **Display Type** to image and regenerating, you will get something like this (do not forget to upload a picture first!):

If the file you uploaded was an audio file (this is possible with the ORDDOC and ORDAUDIO types), you will get something like this:

Suggestion: By default there are limits on the size of the file that can be uploaded. If they are exceeded, you get a Java exception:
`java.io.EOFException: Per-request disk space limits exceeded.`

If you want to change the file size limitations, have a look at the Development Guidelines for Oracle ADF Faces Applications, chapter File Upload, section Configuration, at http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/doc/devguide/fileupload.html - Configuration.

6.10.1. Combining File Display Options

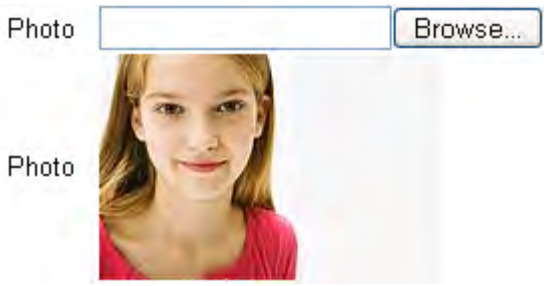
For generating both file upload and file download (or image or audio player) for the same database column, create an extra item as follows:

1. Open the Application Definition Editor.
2. Add an extra item to the group.
3. Copy all properties of the original file item (Photo).
4. Change the **Name** of the new item (for example to ShowPhoto), and change the **Display Type** to fileDownload or image.

General	
Bound to Model Attribute?	☑
Name *	ShowPhoto
Attribute Name	Photo
Value	
Java Type *	OrdImageDomain
Display Type *	image

5. Consider to change the prompt of one or both of the Photo items

If you don't change the prompts, you will get something like this:



6.10.2. Showing Properties of Uploaded Files

When uploading files, several additional characteristics of the files, like size and mime type, are stored in the interMedia database column. You can make these properties visible in your page. What properties are available depends on the object type (ORDDOC, ORDIMAGE or ORDAUDIO).

Reference: For a complete overview of the available properties, see the *inter*Media section of the Oracle Technology Network (http://otn.oracle.com/products/intermedia).

Here are some properties that you might want to show:

Property	SQL name	ADF BC method	Java type
File Size (in bytes)	contentLength	getContentLength()	int
File Mime Type	mimeType	getMimeType()	String
(Original) File Name	source.srcName	getSourceName()	String
File Upload Time	source.updateTime	getUpdateTime()	Timestamp

The SQL name is what you can use to retrieve the property in a SQL query, for example:

```
select emp.photo.contentLength, emp.photo.source.srcName
from   employees emp
where  emp.last_name = 'King';
```

If you want to show some of these additional file properties in your JHeadstart application, for example the File Name and the File Size, here is how you do that.

1. In the ADF BC View Object, create new transient attributes for the File Name and the File Size. See Steps to create a transient attribute.

2. Give the attributes the appropriate types (String and Number).

3. In the View Row Class, change the get methods for the new transient attributes as follows (assuming that the original file attribute is called Photo):

```
public String getPhotoFileName()
{
  String fileName = null;
  if (getPhoto() != null)
  {
    try
    {
      fileName = getPhoto().getSourceName();
    }
    catch (SQLException e)
    {
```

```
      throw new JboException(e);
    }
  }
  return fileName;
}

public Number getPhotoSize()
{
  Number size = null;
  if (getPhoto() != null)
  {
    try
    {
      size = new Number(getPhoto().getContentLength());
    }
    catch (SQLException e)
    {
      throw new JboException(e);
    }
  }
  return size;
}
```

4. Go to the Application Definition, and synchronize the group to get items for the new attributes.

5. Change the item properties (for example the Prompt) where desired, and generate the application.

If you want to use the file name as the label for a download link (in case the file is not an image, video or audio file), you can use the **Hint Text** property on the fileDownload item to refer to the item that holds the name of the file.

For example, if you have a group named "Employees", an item "DocItem" with **Display Type** "fileDownload" that is based on an attribute of type OrdDocDomain, and an item "DocItemFileName" that returns the name of the uploaded DocItem using the technique explained above, you can set the **Hint Text** property of DocItem as follows:

```
#{bindings.EmployeesDocItemFileName.inputValue}
```

This will display the name of the actual file on the download link.

6.10.3. Using JHeadstart File Up/Download on BLOB Columns

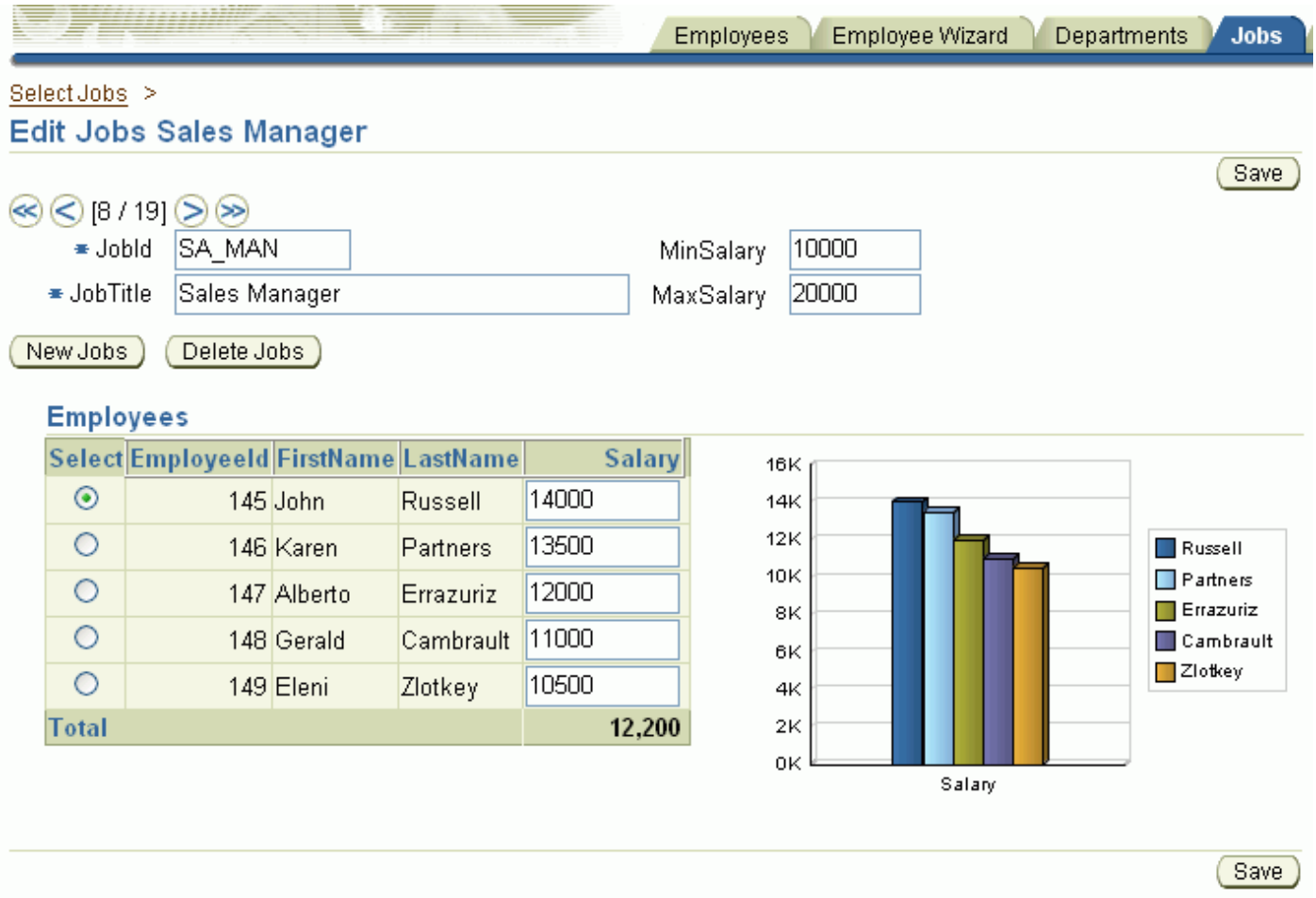
JHeadstart 10.1.2 supported upload and download functionality of files stored as BLOB columns in the database. In JHeadstart 10.1.3 this is not supported anymore because of the superior functionality of the Intermedia object types. If, however you

still have BLOB columns and want to have file upload and download functionality on them, there is a workaround, which involves on-the-fly-creation of ORDDOC objects in a SQL Query on the table with the BLOB column holding the files.

Reference: Lucas Jellema described a workaround on the AMIS Blog: "Enabling BLOB support with JHeadstart - Uploading/Downloading files to and from a BLOB column" at http://technology.amis.nl/blog/?p=2463.

6.11. Generating a Graph

The values that are available in a numeric item can be displayed as a graph, like the example screen shot below:

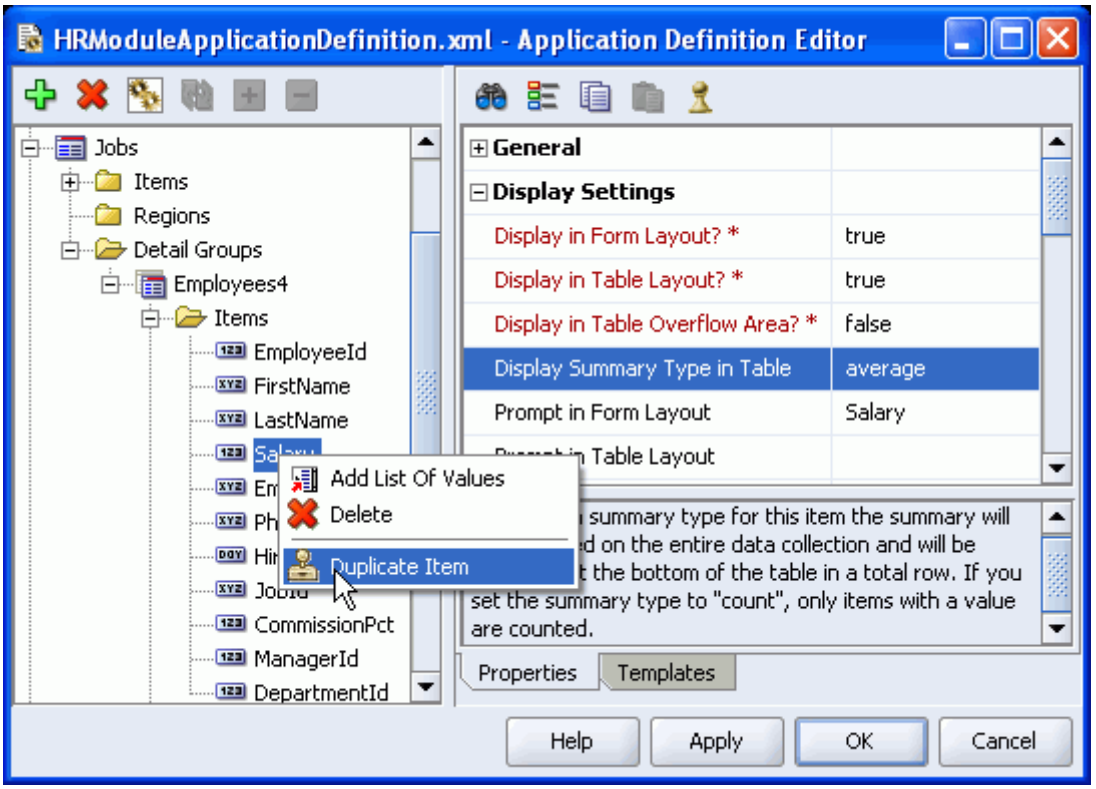


In this example the Salary is both shown in the Employees table, and shown as a graph in the table overflow area. Therefore this example includes making a second Salary item based on the same View Object attribute. By making this second item dependent on the first Salary item, you can change the Salary value and immediately see the change reflected in the graph.

The steps to create such a graph are:

1. In the Employees detail group, set **Layout Style** to table and **Table Overflow Style** to right.

2. Right-mouse-click on the Salary Item and choose Duplicate Item from the popup window.



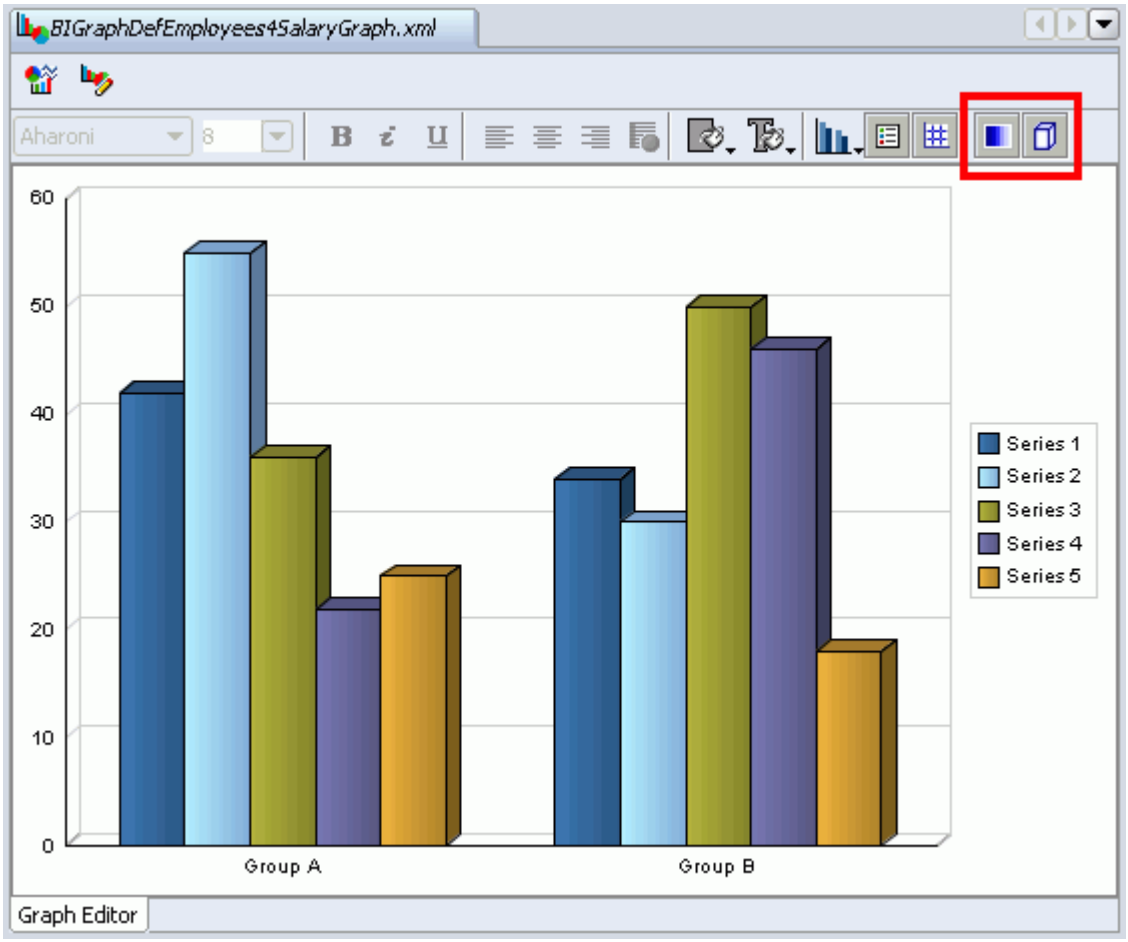
3. A new item named CopyOfSalary is added to the group. Move this new item using drag and drop to display right below the Salary item. Set the following properties for the CopyOfSalary item:

Property Category	**Property Name**	**Set to Value**
General	Name	"SalaryGraph"
General	Display Type	graph
Display Settings	Display in Table Layout	false
Display Settings	Display in Table Overflow Area?	true
Display Settings	Width	"300"
Display Settings	Height	"200"
Display Settings	Depends on Item(s)	Salary

4. Run the JHeadstart Application Generator.

When the generator is finished, you will notice a new file that has been added to your JDeveloper project. This file, named BIGraphDefEmployees4SalaryGraph.xml, is automatically opened in the JDeveloper Graph Editor. Using the Graph Editor, you can set the graph type, for example bar, line or pie chart as well as the visual appearance of the graph. In the example screenshot above, we used the default bar graph, with the visual

appearance of the graph changed to a nice 3D-style rendering of the graph as shown in the picture below.



6.12. Conditionally Dependent Items

The ADF Faces components that JHeadstart application generator uses for your web tier pages cleverly combine Asynchronous JavaScript, XML, and Dynamic HTML to deliver a much more interactive web client interface for your business applications. In ADF Faces, the feature is known as partial page rendering because it allows selective parts of a page to be re-rendered to reflect server-side updates to data, without having to refresh and redraw the entire page. This combination of web technologies for delivering more interactive clients is known more popularly by the acronym AJAX. ADF Faces supports this powerful feature for any JavaServer Faces (JSF) page with no coding. JHeadstart automatically configures the necessary properties on the controls to enable a maximal use of this great feature, for example for enabling dynamically-changing, conditionally-dependent fields.

Sometimes, one field value (or its enabled status or some other characteristic) might depend on another field. JHeadstart makes it simple to generate pages that support this kind of conditionally-dependent field.

In this section we first describe the general usage of the Depends On property, and then build on that for describing how to create cascading lists in form layout, search area, and table layout, where the latter is a special case of row-specific dropdown lists.

6.12.1. Using the Depends On property

For each item, you can specify that it depends on one or more other items in the same group. The details differ a bit if it depends on multiple items as opposed to depending on a single item.

6.12.1.1. If an item depends on a single other item

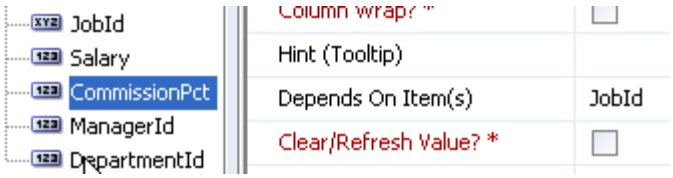
For example, imagine that the commission percentage of an employee only is relevant if they are an Account Manager. In this section we'll configure a simple example to implement the disabling of the CommissionPct item in the Employees group unless the value of the JobId is equal to 'AC_MGR'. To accomplish this task, follow these steps:

- Conditionalize the Value of the Disabled Property Using an Expression: in the JHeadstart Application Definition Editor, expand the top-level Employees group, its Items folder, and select the CommissionPct item. Set its **Disabled?** property to the expression value:

  ```
  #{$DEPENDS_ON_ITEM_VALUE$ != 'AC_MGR'}
  ```

The token $DEPENDS_ON_ITEM_VALUE$ gets substituted by the JHeadstart application generator so that the expression ends up referencing the correct value of the item on which the current item depends. In table layout, you need a different expression than in form layout. We'll setup this item dependency next...

- Set the CommisionPct Item to Depend on the JobId Item by setting its **Depends on Item** property to JobId.



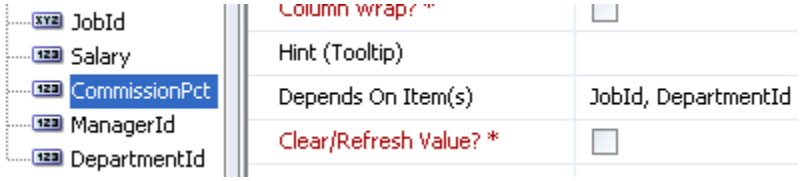
- Choose a value for the **Clear/Refresh Value?** property. When this checkbox is checked, and the depends-on-item changes value, this item's value is cleared *before* the model binding of the depends-on-item is updated. If you would code logic in the setter method of the underlying attribute of the depends-on-item to update this item's value, then this new value will be displayed in the page. If you don't know what to choose, leave it unchecked.

- After regeneration and running the application, in the Employees tab, if you use the Quick Search area to find all employees whose LastName starts with the letter H, and then drill down to the details, you can navigate between employees like Michael Hartstein and Shelly Higgins to notice that the CommissionPct field on the screen as disabled for Michael, as shown below, but enabled for Shelly (whose JobId = 'AC_MGR').

6.12.1.2. If an item depends on multiple other items

The basic steps are the same as when the item depends on a single item, except for the following:

- Instead of choosing the Depends On item from the dropdown list, type in a comma-separated list of item names in the field, followed by using the Enter key.



- You cannot use the token $DEPENDS_ON_ITEM_VALUE$ if the item depends on multiple other items. Instead, use the following expressions depending on the layout style and search area of the group:

Item usage	**Expression**
Table layout	row.<attributeName>
Form layout	bindings.<groupName><itemName>.inputValue
Search area	search<groupName>.criteria.<groupName><itemName>

  For example, to refer to the value of the JobId in the search area, use the expression `#{searchEmployees.criteria.EmployeesJobId}`.

- If in your application you have more than one of the abovementioned item usages, for example you have the dependency in both form layout and search area, you will have to create multiple dependent items: one for each usage. Make sure the copied dependent item is displayed only in table layout, or only in the form layout, or only in the search area, and use the appropriate expression for each copied item.

6.12.2. Cascading Lists

If the displayed values in a dropdown list depend on the chosen value in another dropdown list, we call them cascading lists.

Here are the basic steps to generate this in a form layout using the Region-Countries example from the HR schema:

- Create a ViewObject on Countries with a where clause named bind param: region_id = :p_region_id
- define p_region_id on the Bind Params tab of your VO
- Set the **Query Bind Parameter** property in the dynamic domain created for the country item drop down list according to the table below, for example: `p_region_id=#{bindings.CountriesRegionId.inputValue}`

Item usage	**Expression**
Table layout	row.<attributeName>
Form layout	bindings.<groupName><itemName>.inputValue
Search area	search<groupName>.criteria.<groupName><itemName>

Attention: If you have the same cascading lists in table and form layout, and/or in a search area, you need to make separate domains, and separate items for CountryId: one displayed in table layout with the "table" domain associated, one in form layout with the "form" domain associated, one in search are with the "search area" domain associated.

- In case of a *table* layout the domain checkbox **Data Collection Changes By Row** must be checked as well.
- Set **Depends On Item(s)** for CountryId item to the RegionId item
- Check checkbox **Clear/Refresh Value?** for CountryId
- Generate and run.

6.12.3. Row Specific Dropdown Lists in Table

This is in fact a more generic case of Cascading Lists in Table Layout, so follow the steps above!

6.13. Custom Button that Calls a Custom Business Method

Often you will have special processing that you want to initiate by clicking a button. It could be a stored procedure in the database, or a custom method in ADF Business Components.

Attention: This section assumes you understand the concept of JHeadstart Generator Templates and you know how to use custom templates as explained in section 4.7

Reference: See also section 8.5.4.1 "Add Commit Behavior to a Custom Button".

The example below calls a custom ADF BC Application Module method from a button in a JSF page. If you want to call a stored PL/SQL procedure or function, let this custom method call the stored function or procedure like is described in the ADF Developer's Guide for Forms/4GL Developers.

Reference: See the Oracle Application Developer Framework Developer's Guide for Forms/4GL Developers Release 10.1.3, section 25.5: Invoking Stored Procedures and Functions.

In this example we will create a group level button to increase the salary of the current employee. We discuss two ways:

1. Generating just a button that increases the salary with a fixed percentage.
2. Generating a button with an input field to type in the desired percentage.

In both cases we need a custom method in our Application Module.

6.13.1. Creating a Custom Method in the ADF BC Application Module

We are going to use the same method for both examples, which takes the employee id and the desired percentage as parameters.

- Go to the Model project, and find the Application Module (for example HRService).
- Go to the Application Module Class (for example HRServiceImpl.java). If there is none, generate one using the Application Module editor, Java category.
- Create a new method as follows (assuming that the EmployeesView is defined in the Application Module's data model as EmployeesView1):

```
public void increaseSalary(String empId, String percentage)
{
  // 1. Find the Employees view object instance
  EmployeesViewImpl empView = getEmployeesView1();
  // 2. Construct a new Key to find the Employee with the specified id
  Key empKey = new Key(new Object[]
      { empId });
  // 3. Find the row matching this key
  Row[] empsFound = empView.findByKey(empKey, 1);
  if (empsFound != null && empsFound.length > 0)
  {
    EmployeesViewRowImpl employee = (EmployeesViewRowImpl) empsFound[0];
    // 4. Increase the salary with the specified percentage
```

```
        double convertedPercentage = new Double(percentage).doubleValue();
        double multiplyNumber = 1 + convertedPercentage / 100;
        Number newSalary = employee.getSalary().multiply(multiplyNumber);
        employee.setSalary(newSalary);
      }
    }
```

Attention: The type of the parameters is String, because that is easiest to pass from the ViewController, and it can be converted to any type you need.

Attention: This code assumes that you created a View Row class for the Employees view. You can generate this class by going to the EmployeesView editor, selecting the Java category and checking the EmployeesViewRowImpl class checkbox.

Attention: The Number class referenced is `oracle.jbo.domain.Number`, not `java.lang.Number`.

- Publish this method to the ADF Data Control by opening the Application Module Editor, and going to the Client Interface category. Shuttle the new increaseSalary method to the right.

Now you can choose between two customizations:

1. Call the increaseSalary method from a button where you pass the current Employee Id and a fixed percentage of 10.

2. Call the increaseSalary method from a button where you pass the current Employee Id and take the percentage from an input field on the page.

If you choose 1, apply the steps from section Creating a Button that Calls the Method With a Fixed Percentage, and if you choose 2, apply the steps from section Creating a Button that Calls the Method With Percentage From Input Field.

6.13.2. Creating a Button that Calls the Method With a Fixed Percentage

Before we generate the button into the page, we drag-and-drop it using the JDeveloper Visual Editor, including the relevant ADF Bindings. That way we can see what it is we need to generate.

Before making any customizations, ensure that the names of the used generator templates are included in the sources, as explained in the section Finding Out Which Generator Templates Are Used.

- Go to the page where you want to add the button (for example Employees.jspx) in Design mode.

- Go to the Data Control Palette, and find the new increaseSalary method of the Application Module.

- Drag-and-drop the increaseSalary method next to the other group level buttons.

- Choose Create – Methods – ADF Command Button. An Action Binding Editor dialog opens. Don't close it yet.

We will now pass the current Employee Id, and a fixed value of 10 (to increase the salary with 10 percent) to the increaseSalary method.

- Double click the empty value cell of empId, click the Button with three dots (Edit), and choose ADF Bindings – bindings – EmployeesEmployeeId - inputValue. Click the button with the right arrow (>) and the expression `${bindings.EmployeesEmployeeId.inputValue}` will appear. Click OK.

- Double click the empty value cell of percentage, click the Button with three dots (Edit), and type in the expression 10. Click OK.

- Click OK on the Action Binding Editor.

This creates a button 'increaseSalary' in the page, and a binding 'increaseSalary' in the Page Definition of the page. If you run the page, and click the button, you can see a higher value in the Salary field.

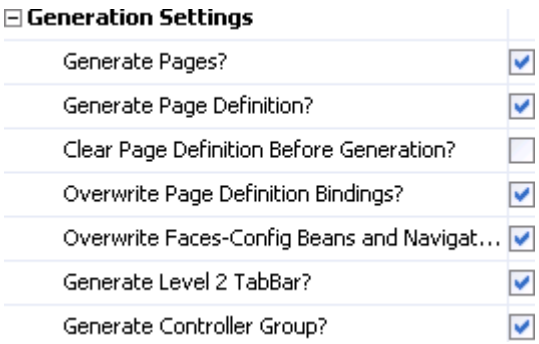
But if you now run the JHeadstart generator, you will lose the button and the binding, so we must find a way to generate them.

6.13.3. Generating the Button that Calls the Method

We want JHeadstart to leave the increaseSalary binding in the Page Definition, so we are going to instruct the generator only to overwrite its own JHeadstart bindings and leave other bindings alone.

- Go to the Application Definition Editor, and switch to Expert mode (see section Expert mode). This will ensure that the Generation Settings property category becomes visible at group level (between Deep Linking and Customization Settings).

- Go to the Employees group, and uncheck the **Clear Page Definition Before Generation?** property.



To get the increaseSalary button in the generated JSF page, we are going to customize one of JHeadstart's generator templates. First we need to find out which template this is. Before making any customizations, ensure that the names of the used generator templates are included in the sources, as explained in the section Finding Out Which Generator Templates Are Used.

Then you can open the source of the generated Employees.jspx, and look for increaseSalary. There is your increaseSalary button in between the comments explaining which template generated what:

```
<!-- DEBUG:BEGIN:FORM_PAGE_CONTENT : default/page/formPageContent.vm, nesting level: 1 -->
<f:facet name="actions">
  <af:panelButtonBar id="pageButtons">

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

<!-- DEBUG:BEGIN:NEW_BUTTON : default/button/newButton.vm, nesting level: 2 -->
  <af:commandButton actionListener="#{bindings.CreateEmployees.execute}"
                    action="CreateEmployees"
                    textAndAccessKey="#{nls['NEW_BUTTON_LABEL_EMPLOYEES']}"
                    rendered="#{!createModes.CreateEmployees  }"
                    immediate="true"
                    onclick="return alertForChanges();"
                    id="EmployeesNewButton">
    <f:actionListener type="oracle.jheadstart.controller.jsf.listener.DoRollbackActionListener"/>
    <af:resetActionListener/>
      </af:commandButton>
<!-- DEBUG:END:NEW_BUTTON : default/button/newButton.vm, nesting level: 2-->



<!-- DEBUG:BEGIN:DELETE_BUTTON : default/button/deleteButton.vm, nesting level: 2 -->
  <af:commandButton actionListener="#{bindings.DeleteEmployees.execute}"
                    action="DeleteEmployees"
```

As you can see, the buttons are included within the actions facet that is printed by the FORM_PAGE_CONTENT template, which defaults to default/page/formPageContent.vm. So, let's customize this template and add the custom button to it.

Attention: Group-level buttons are not always generated through the FORM_PAGE_CONTENT template. This template is used when there is only one group displayed on the page, in which case the group-level buttons are printed on the same level as the page level Save button. When multiple groups are displayed on a page, the FORM_GROUP_BUTTONS or TABLE_GROUP_BUTTONS templates are used, depending on the group layout style.

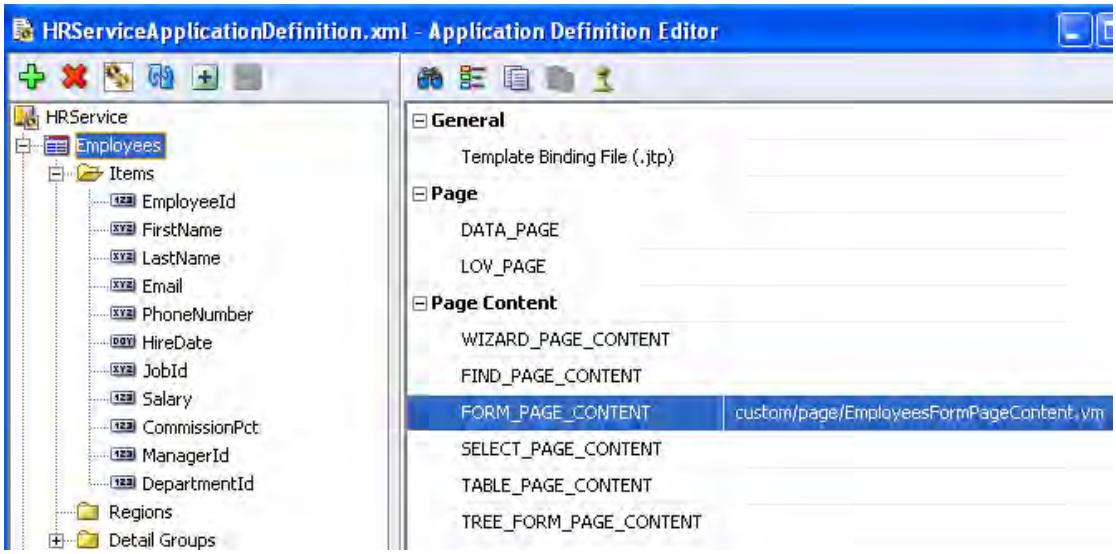
- Open default/page/**formPageContent.vm** and Save As **custom**/page/**Employees**FormPageContent.vm.
- In the `<f:facet name="actions">` in the template, locate the place where you want to insert the custom button (before, in between, or after the other buttons).
- Go to the Employees.jspx page, and copy the `<commandButton>` section of the custom button. Paste it in the desired location in the panelButtonBar of EmployeesFormPageContent.vm.

EmployeesFormPageContent.vm

```
1  <f:facet name="actions">
2    <af:panelButtonBar id="pageButtons">
3  ## customization: added increase salary button
4      <af:commandButton actionListener="#{bindings.increaseSalary.execute}"
5                        text="Increase Salary 10%"
6                        disabled="#{!bindings.increaseSalary.enabled}"/>
7  ## end of customization
8      ## render buttons to go to detail group for level 2 and lower
```

Now the only thing left to do is make sure that the customized template is used for the Employees group.

- Open the Application Definition Editor, go to the Employees group, and click on the Templates tab.
- Set the FORM_PAGE_CONTENT value to custom/page/EmployeesFormPageContent.vm



If you now run the JHeadstart Application Generator again, the button is included in the generated page, and the binding for the button is preserved.

```
<!-- DEBUG:BEGIN:FORM_PAGE_CONTENT : custom/page/EmployeesFormPageContent.vm, nesting level: 1 -->
<f:facet name="actions">
  <af:panelButtonBar id="pageButtons">
    <af:commandButton actionListener="#{bindings.increaseSalary.execute}"
                      text="Increase Salary 10%"
                      disabled="#{!bindings.increaseSalary.enabled}"/>
```

6.13.4. Creating a Button that Calls the Method With Percentage From Input Field

Before we generate the button and the percentage input field into the page, we drag-and-drop it using the JDeveloper Visual Editor, including the relevant ADF Bindings. That way we can see what it is we need to generate.

Before making any customizations, ensure that the names of the used generator templates are included in the sources, as explained in the section Finding Out Which Generator Templates Are Used.

- Go to the page where you want to add the button and input field (for example Employees.jspx) in Design mode.
- Go to the Data Control Palette, and find the new increaseSalary method of the Application Module.
- Drag-and-drop the increaseSalary method just below the Employee fields.
- Choose Create – Parameters – ADF Parameter Form. A Form Fields Editor dialog opens. Don't close it yet.

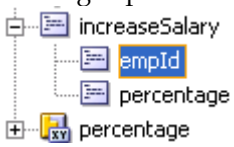
We only want a Form Field for the percentage, as we will pass the current Employee Id to the increaseSalary method behind the scenes.

- By default the empId row is highlighted, so you can click the Delete button right away.
- Click OK on the Form Fields Editor.

This creates a form with an input field and a button 'increaseSalary' in the page. In the Page Definition of the page it creates a variableIterator with a variable 'increaseSalary_percentage', an attribute binding 'percentage', and a method binding 'increaseSalary'.

Now we still have to pass the Employee Id to the method call.

- Go to the Page Definition of the Employees page.
- In the Structure pane, open the bindings node and find the increaseSalary binding. Open that node too, and select the empId parameter.



- Go to the properties of this parameter, and find the NDValue property. Click the Button with three dots (Edit), and choose ADF Bindings – bindings – EmployeesEmployeeId - inputValue. Click the button with the right arrow (>) and the expression `${bindings.EmployeesEmployeeId.inputValue}` will appear. Click OK and Save.

If you run the page, fill in a percentage, and click the button, you can see a higher value in the Salary field. If you like you can improve the layout by replacing the panelForm by a panelLabelAndMessage (which means that you have to set simple="true" on the inputText component).

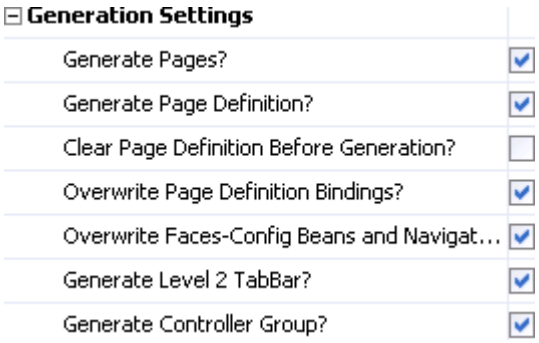
But if you now run the JHeadstart generator, you will lose the new page components and bindings, so we must find a way to generate them.

6.13.5. Generating the Input Field and Button that Calls the Method

We want JHeadstart to leave the increaseSalary and related bindings in the Page Definition, so we are going to instruct the generator only to overwrite its own JHeadstart bindings and leave other bindings alone.

- Go to the Application Definition Editor, and switch to Expert mode (see section Expert mode). This will ensure that the Generation Settings property category becomes visible at group level (between Deep Linking and Customization Settings).

- Go to the Employees group, and uncheck the **Clear Page Definition Before Generation?** property.



To get the percentage input field and the increaseSalary button in the generated JSF page, we are going to customize one of JHeadstart's generator templates. First we need to find out which template this is. Before making any customizations, ensure that the names of the used generator templates are included in the sources, as explained in the section Finding Out Which Generator Templates Are Used.

Then you can open the source of the generated Employees.jspx, and look for increaseSalary. There is your increaseSalary button with percentage input field in between the comments explaining which template generated what:

```
</af:panelGroup>

<!-- DEBUG:END:FORM_GROUP : default/pageComponent/formGroup.vm, nesting level: 2-->


<!-- DEBUG:END:FORM_PAGE_CONTENT : default/page/formPageContent.vm, nesting level: 1-->
            <af:panelForm>
              <af:inputText value="#{bindings.percentage.inputValue}"
                            label="#{bindings.percentage.label}"
                            required="#{bindings.percentage.mandatory}"
                            columns="#{bindings.percentage.displayWidth}">
                <af:validator binding="#{bindings.percentage.validator}"/>
              </af:inputText>
              <af:commandButton actionListener="#{bindings.increaseSalary.execute}"
                                text="increaseSalary"
                                disabled="#{!bindings.increaseSalary.enabled}"/>
            </af:panelForm>
          </af:panelPage>
```

As you can see, the inputText and commandButton are included between the end of the panelGroup that is printed by the FORM_GROUP template and the end of the panelPage that is printed by the DATA_PAGE template (the starting point for all templates). Let's customize the FORM_GROUP template, which defaults to default/pageComponent/formGroup.vm.

- Open default/pageComponent/**formGroup.vm** and Save As **custom**/pageComponent/**Employees**FormGroup.vm.
- Locate the place where you want to insert the custom button (an appropriate place seems to be after the end of the first panelGroup, before the FORM_GROUP_BUTTONS).
- Go to the Employees.jspx page, and copy the `<panelForm>` section with all its child components. Paste it in the desired location of EmployeesFormGroup.vm.

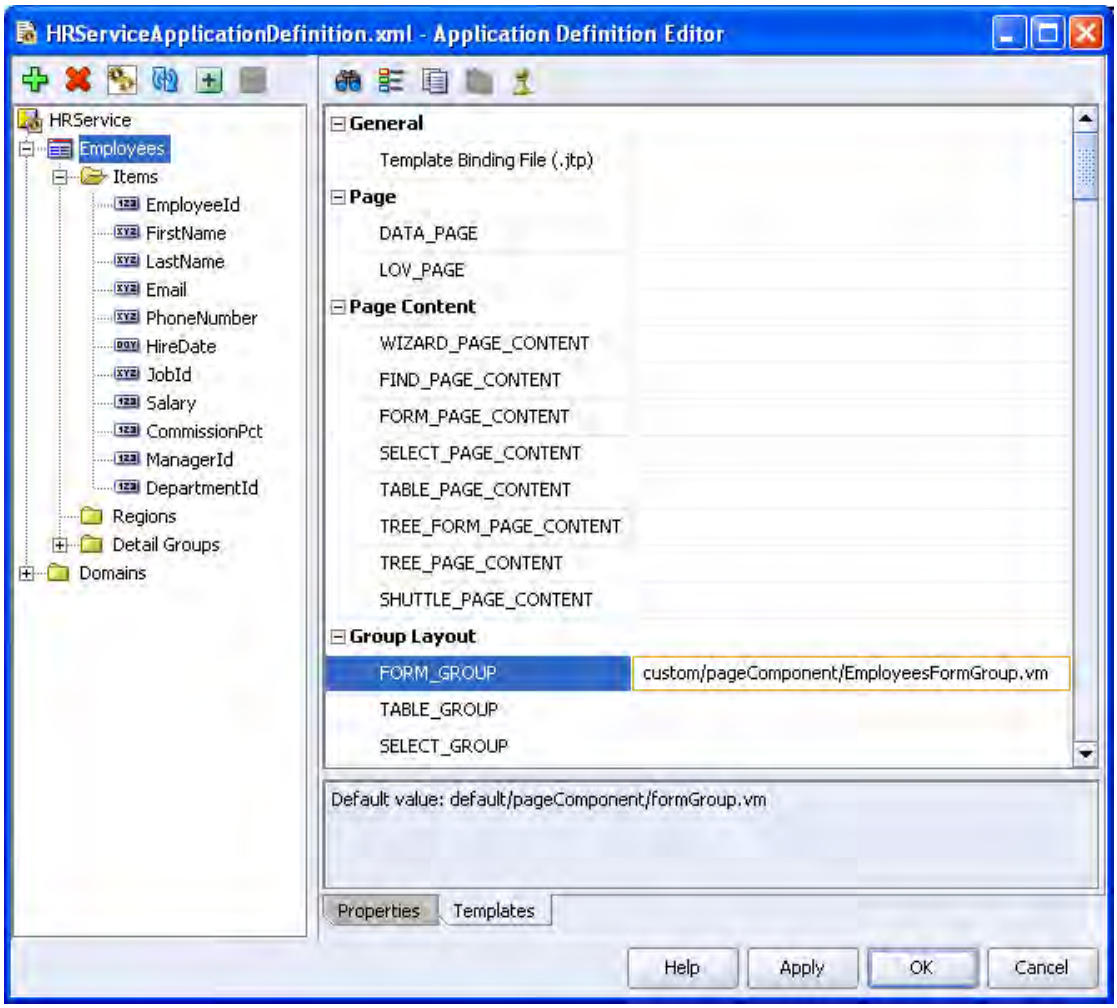
EmployeesFormGroup.vm

```
26    </af:panelGroup>
27
28  ## customization: added increaseSalary button with percentage input field
29    <af:panelLabelAndMessage label="Percentage">
30      <af:inputText value="#{bindings.percentage.inputValue}"
31                    simple = "true"
32                    required="#{bindings.percentage.mandatory}"
33                    columns="#{bindings.percentage.displayWidth}">
34        <af:validator binding="#{bindings.percentage.validator}"/>
35      </af:inputText>
36      <af:commandButton actionListener="#{bindings.increaseSalary.execute}"
37                        text="Increase Salary"
38                        disabled="#{!bindings.increaseSalary.enabled}"/>
39    </af:panelLabelAndMessage>
40  ## end of customization
41
42    #JHS_PARSE("FORM_GROUP_BUTTONS" ${JHS.current.model})
43
44    <af:objectSpacer height="10" id="${JHS.current.group.shortName}FormGroupSpacer"/>
```

Now the only thing left to do is make sure that the customized template is used for the Employees group.

- Open the Application Definition Editor, go to the Employees group, and click on the Templates tab.

- Set the FORM_GROUP value to custom/pageComponent/EmployeesFormGroup.vm



If you now run the JHeadstart Application Generator again, the input field and button is included in the generated page, and the bindings for the button are preserved.

6.14. Hyperlink to Navigate Context-Sensitive to Another Page (Deep Linking)

JHeadstart allows you generate hyperlinks (or buttons) that navigate context-sensitive to another page. With context-sensitive we mean that the data displayed in the target page, depends on the data displayed in the source page. This feature is called deep linking and can be implemented using custom generator templates.

Attention: This section assumes you understand the concept of JHeadstart Generator Templates and you know how to use custom templates as explained in section 4.7

In this example, we will use the JHeadstart deep linking support in combination with a custom template to generate the JobId in the Employees group table as a hyperlink that navigates to the Job Edit page, querying the proper Job row.

First we must enable deep linking for the Job Edit Page, which is the target of the deep link.

6.14.1.1. Enabling Deep Linking for a Group

To allow creation of deep links to the form page of a certain group (in this example the Jobs group), perform the following steps:

1. In the Application Definition Editor, switch to Expert mode (see section Expert mode). This will ensure that the Deep Linking property category becomes visible at group level (between Table Layout and Generation Settings).

2. Go to the Jobs group and set the **Deep Linking Type** to Query By Key Value.

3. Set the **Deep Linking Key Expression** to `#{param.jobId}`

Deep Linking	
Type of Deep Linking	Query By Key Value
Enable Deep Linking Expression	#{jsfNavigationOutcome=='DeepLink$GROUP_NAME$'}
Deep Linking Key Expression	#{param.jobId}
Generation Settings	

4. Generate the application, to make a navigation rule DeepLinkJobs available (it is needed in the following steps).

These deep linking settings ensure that when a JSF navigation outcome called 'DeepLinkJobs' is used (this condition is specified in the **Enable Deep Linking Expression**), it will go to the Jobs page, and query the Jobs data collection using the key specified in the request parameter 'jobId' (the **Deep Linking Key Expression**).

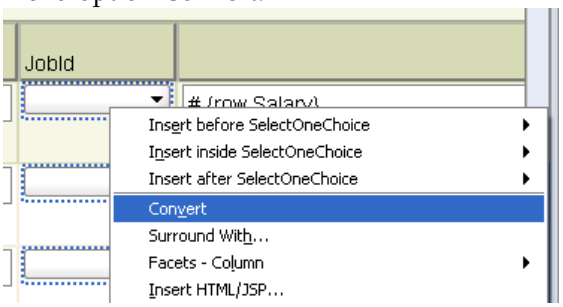
Attention: For the **Type of Deep Linking** you can also choose "Set Current Row Using Key Value". The difference is that instead of requerying the database and returning just the desired row, the current row indicator in the View Objects's row set will be pointed to the desired row.

The next step is to manually change the Job field in the Employees page to a hyperlink, before moving that customization to a generator template.

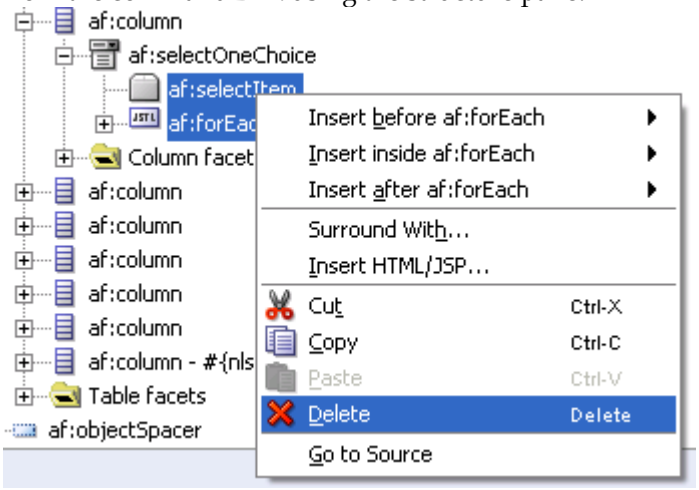
6.14.1.2. Manually Changing a Dropdown List to a Deep Link

1. Open the EmployeesTable.jspx in the Visual Design Editor in JDeveloper.

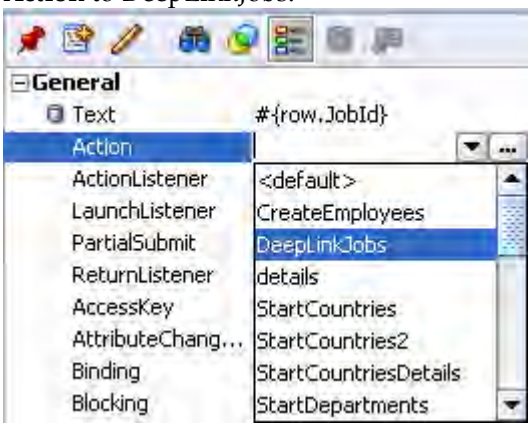
2. Convert the JobId selectOneChoice into a commandLink using the right-mous-menu option Convert.



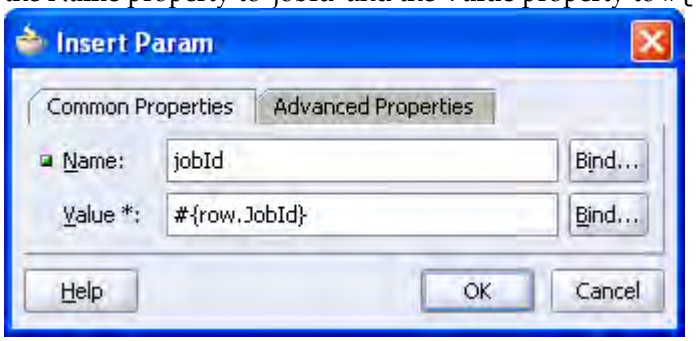
3. Remove the 'left over' child components from the converted selectOneChoice from the commandLink using the structure pane.



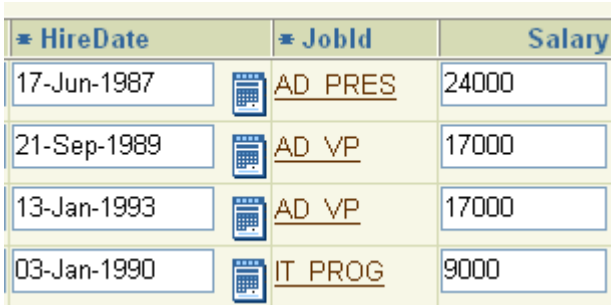
4. Go to the properties of the commandLink, set **Text** to `#{row.JobId}`, and set **Action** to DeepLinkJobs.

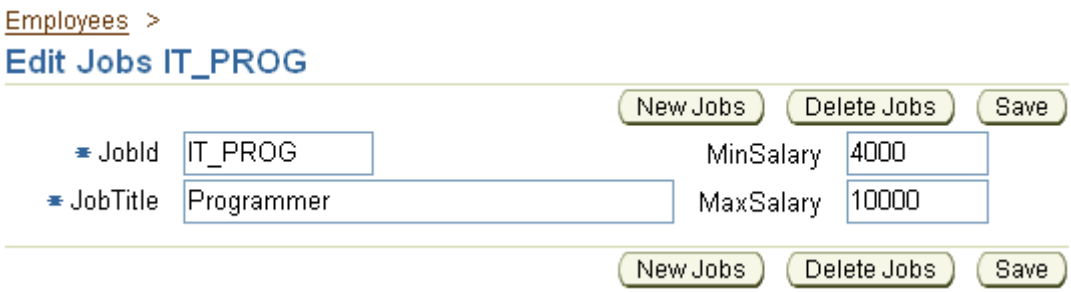


5. Insert inside the commandLink component the JSF Core component 'param'. Set the **Name** property to 'jobId' and the **Value** property to `#{row.JobId}`.



You can now run the EmployeesTable page to test if the Job deep link works correctly.





Note that the breadcrumb on top of the Job Edit page shows that we just came from the Employees page.

6.14.1.3. Preserving the Manual Changes using an Item Template

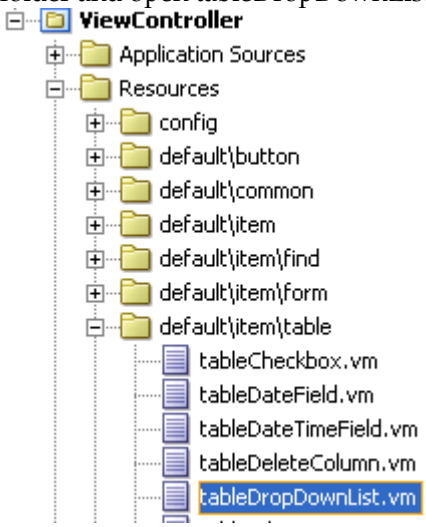
If we would now run the JHeadstart Application Generator, it would overwrite the manual changes we made in the Employees page. Therefore we will create a Generator Template for the JobId item to preserve the changes.

JHeadstart would generate a dropdown list for the JobId item in the table page, so the template that we need to customize is TABLE_DROP_DOWN_LIST. See also section Finding Out Which Generator Templates Are Used.

1. In the source of the EmployeesTable.jspx file, copy the commandLink element.

```
<af:commandLink id="EmployeesJobId"
                text="#{row.JobId}"
                action="DeepLinkJobs">
  <f:param name="jobId" value="#{row.JobId}"/>
</af:commandLink>
```

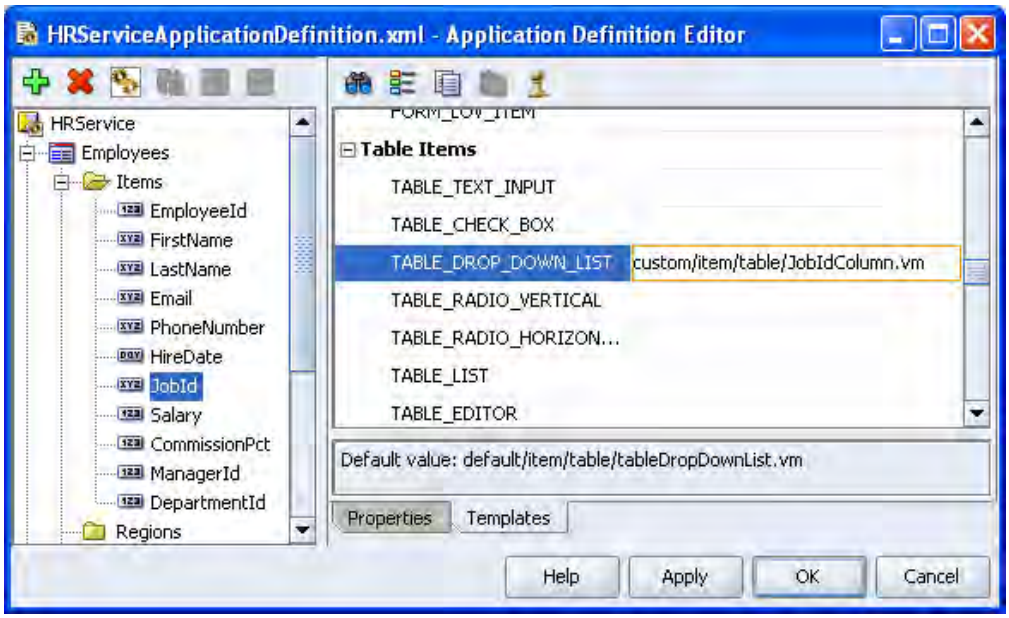
2. In the JDeveloper navigator, go to the project's Resources – default\item\table folder and open tableDropDownList.vm.



3. Save it as custom/item/table/JobIdColumn.vm.

4. In JobIdColumn.vm, replace the selectOneChoice component with the commandLink you just copied.

5. Select the JobId item of the Employees group in the Application Definition Editor, and click the Templates tab. Set the TABLE_DROP_DOWN_LIST property to custom/item/table/JobIdColumn.vm.



Now you can safely generate the application again, the Job deep link will still work!

6.15. Embedding Oracle Forms in JSF Pages

OraFormsFaces™ is a JSF component library to integrate Oracle Forms in a JSF web application. This allows a developer to embed Oracle Forms in a JSF page and truly integrate the two, including passing context, events, eliminating Forms applet startup time, and many more features.

OraFormsFaces allows organizations to use the Java stack for new developments while protecting their investment in Oracle Forms. They can build new JSF or ADF Faces based web applications and integrate existing Forms applications in them. The JSF web application can pass parameters to Forms and the other way around. Both Forms and JSF can raise events (commands or triggers) in the other technology.

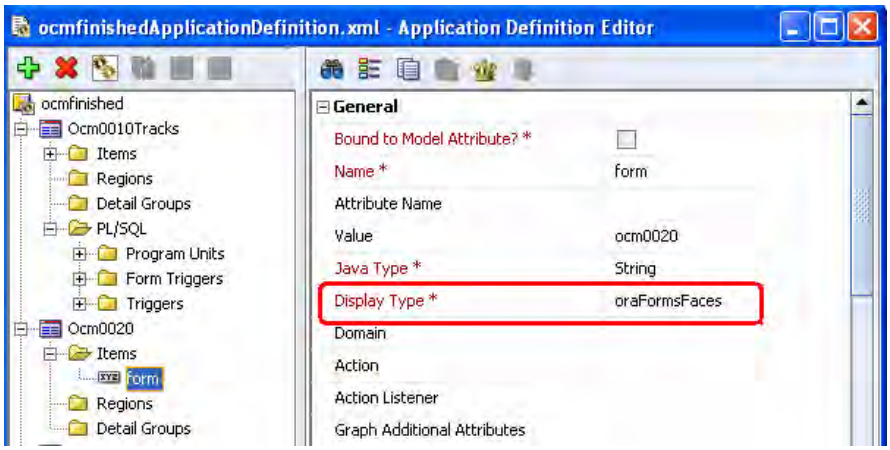
OraFormsFaces is a product from Commit Consulting. A trial version can be downloaded from the Commit Consulting website.

Commit Consulting: For more information on OraFormsFaces and Commit Consulting, go to http://www.commit-consulting.com/
A special OraFormsFaces page for JHeadstart users is also available: http://www.commit-consulting.com/jhs

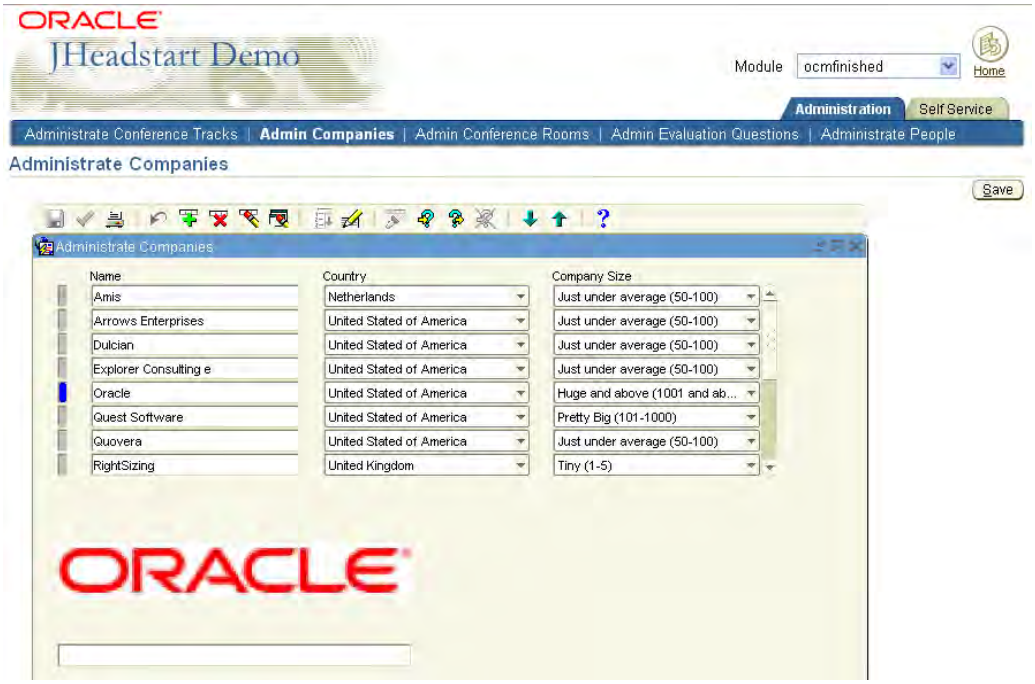
JHeadstart integrates with OraFormsFaces through the item display type "oraFormsFaces".

Follow these steps to generate a web page that embeds an Oracle Form using the OraFormsFaces technology:

- Install OraFormsFaces in JDeveloper, by following the instructions in the OraFormsFaces Developer's Guide.
- Add the OraFormsFaces environment entries in the web.xml of your project, as documented in the OraFormsFaces Developer's Guide.
- Create a new top-level group in the application definition editor. Uncheck the checkbox property **Bound to Model Data Collection**.
- Add an item to the group, uncheck the item checkbox property **Bound to Model Attribute?**
- Set the **Java Type** property to "String"
- Set the **Value** property to the name of the Oracle Form you want to embed.



- Set the **Display Type** property to oraFormsFaces.
- Generate the application
- Make sure the forms server is running.
- Run the generated application. When you go to the menu tab for the group you just created, you will see the Oracle Form embedded in your JSF page. The first time you access the page, it takes a few seconds because the forms applet must be started. When you later return to the page, you will notice the form will be displayed immediately, since the Forms applet is only started once (one of the many features of the OraFormsFaces library). When you click on the JSF Save button, you will notice that the Oracle Form is committed!



CHAPTER

7

Generating Query Behaviors

This chapter discusses how to add query behaviors to a web page. Topics discussed include:

- Using auto-query
- Using Query Bind Parameters
- Quick Search
- Advanced Search
- Forcing a Requery

7.1. Configuring the Query

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

7.1.1. Specifying Auto Query

By default, JHeadstart generates 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 a group 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.

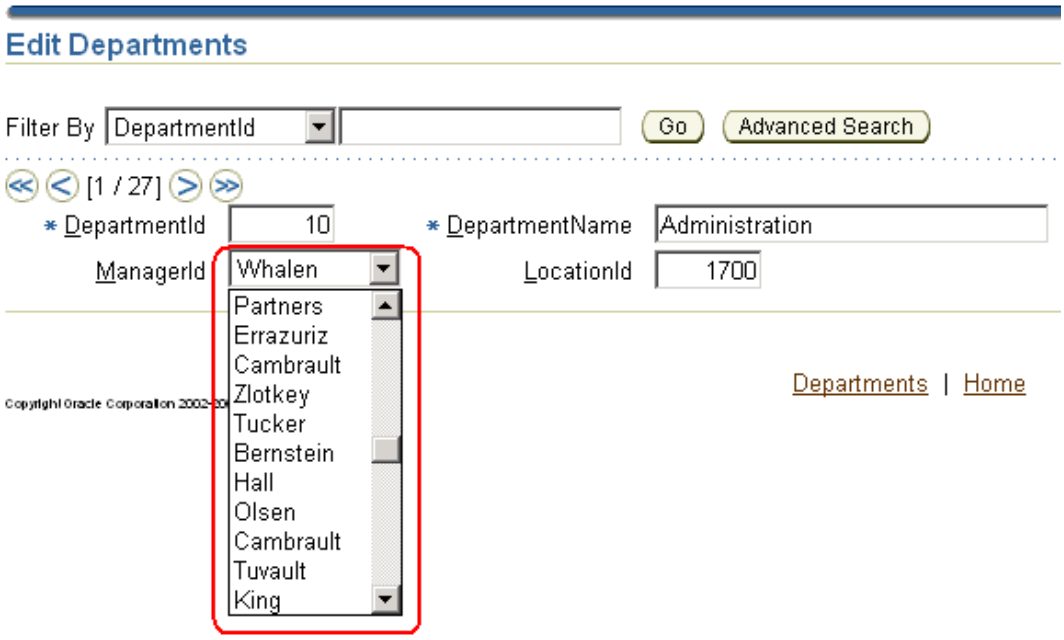
7.1.2. Using Query Bind Parameters

Both Groups and List of Values (indirectly) 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 **Display Type** of the item ManagerId is 'dropDownList'. JHeadstart generates this for us:

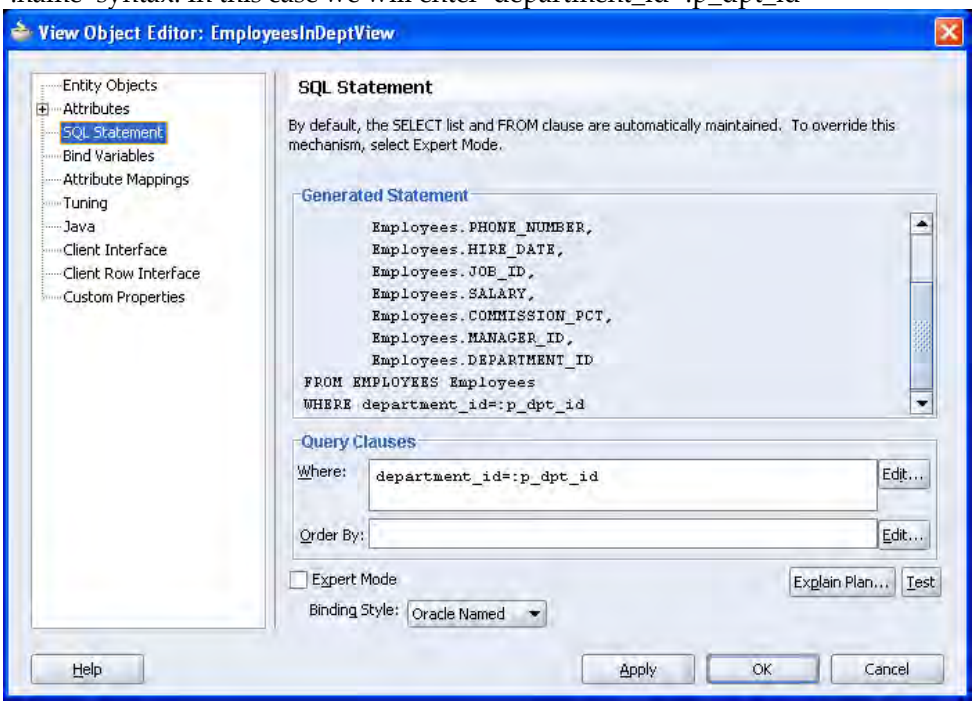


In the dropdown list, all the employees are shown. This is not what we want. We want the manager to be an employee of the department. The dropdown list should only contain employees that are in the department we are maintaining, in this case the department with DepartmentId=10.

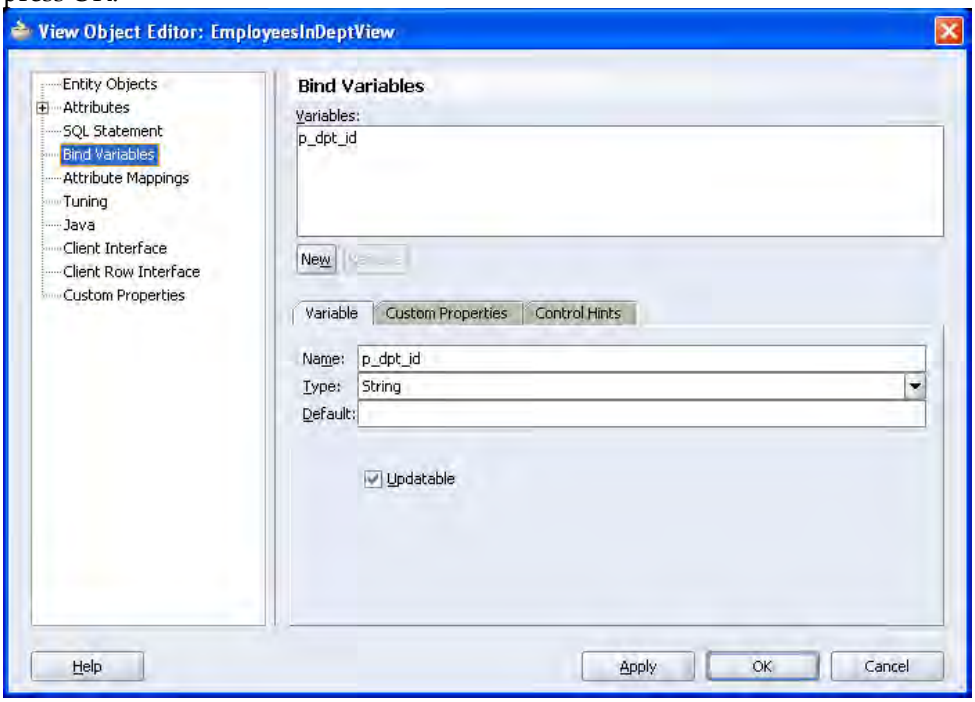
We will implement this requirement by using the Query Bind Parameters of JHeadstart:

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

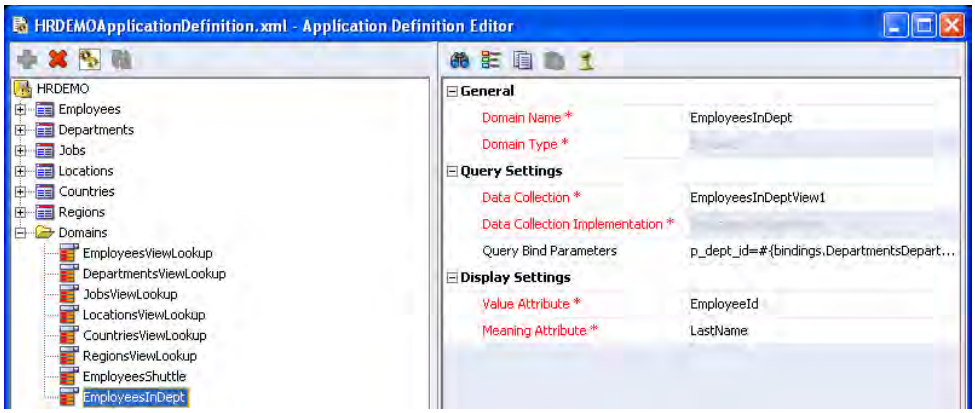
':name' syntax. In this case we will enter 'department_id=:p_dpt_id'



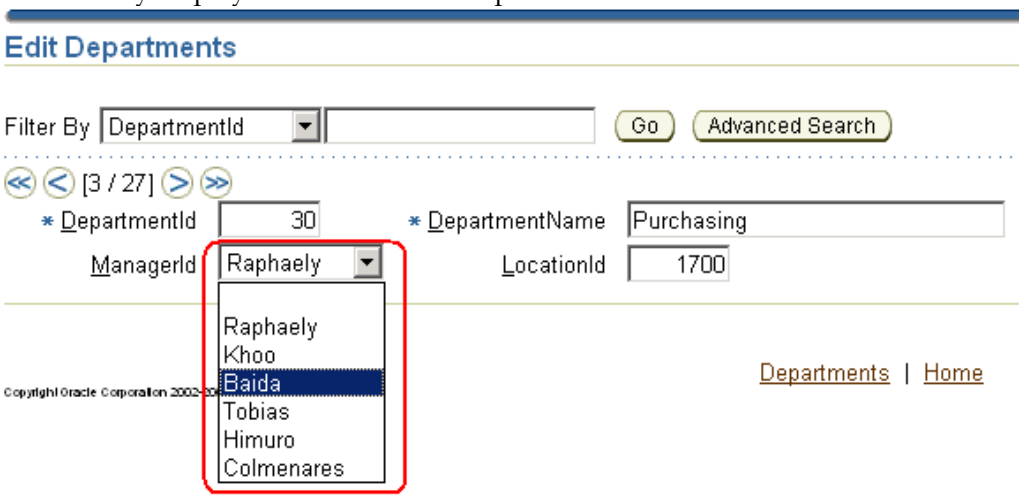
3. Add the new variable in the Bind Variables tab. Enter a name and a type and press OK.



4. Add the new View Object to the Data Model of the Application Module
5. In JHeadstart, create a new Dynamic Domain based on the new View Object and name it something like 'EmployeesInDept'. In the **Query Bind Parameters** property enter this expression:
`p_dpt_id=#{bindings.DepartmentsDepartmentId.inputValue}`



6. Set the property **Domain** (on the ManagerId item) to EmployeesInDept.
7. Generate and run the application again. The dropdown list for ManagerId now contains only employees of the selected department.



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

7.1.3. JHeadstart Runtime Implementation of Query Bind Parameters

When you specify query bind parameters for a group in the Application Definition, JHeadstart generates a QueryBindParams managed bean in the group faces-config. If you specify query bind parameters for a dynamic domain in the Application Definition, JHeadstart generates such a managed bean in the domains faces-config. Here is an example:

```
<managed-bean>
  <managed-bean-name>CountriesViewLookupQueryBindParams</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.bean.QueryBindParams
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>namedParams</property-name>
    <map-entries>
      <map-entry>
        <key>p_region_id</key>
        <value>#{bindings.MyGroupRegionId.inputValue}</value>
      </map-entry>
    </map-entries>
  </managed-property>
</managed-bean>
```

The only task of the `QueryBindParams` class is to hold the last value of the query bind parameters.

Applying the bind parameters is done by method `applyBindParams()` in `JhsApplicationModuleImpl`. This method compares the old and new values of the bind parameters and only (re-)executes a query when at least one bind parameter value has changed.

In the page definition, an action binding is generated to call the `applyBindParams()` method:

```
<methodAction RequiresUpdateModel="true" Action="999"
              id="applyBindParamsCountriesViewLookup"
              DataControl="HRServiceDataControl"
              InstanceName="HRServiceDataControl.dataProvider"
              MethodName="applyBindParams"
              ReturnName="HRServiceDataControl.methodResults.HRServiceDat
              IsViewObjectMethod="false">
  <NamedData NDName="voUsage" NDValue="CountriesViewLookup"
             NDType="java.lang.String"/>
  <NamedData NDName="args"
             NDValue="#{CountriesViewLookupQueryBindParams.namedParams}"
             NDType="java.util.HashMap"/>
</methodAction>
```

Note the EL expression for the second method argument that references the `namedParams` property of the `queryBindParams` bean.

The missing link is how this method action is invoked. The query must be executed before the page is displayed, so we cannot call the method using a button. As explained in chapter "JSF-ADF Page Lifecycle" we can use the invoke action executable for this purpose. JHeadstart generates an invokeAction executable as follows:

```
<executables>
  <invokeAction id="applyBindParamsCountriesViewLookupInvoke"
                Binds="applyBindParamsCountriesViewLookup"
                Refresh="renderModel"/>
```

Note that because of the value of the refresh property, the action binding is invoked in the Prepare Render phase. If we had left the refresh property to its default, the action binding would be invoked twice: once in the Prepare Model phase, and once in the Prepare Render phase. We invoke the action binding in Prepare Render phase to be sure we have the latest up-to-date values of the query bind parameters, they might have been changed in the Update Model or Invoke Application phases.

Reference: See the Javadoc or source of `JhsApplicationModuleImpl.applyBindParams()`.

7.2. 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.

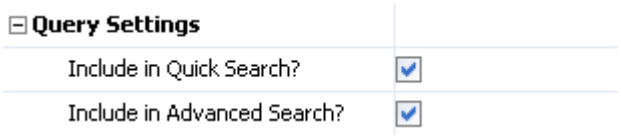
Before generating a Quick Search or Advanced Search page, you have to make some preparations:

7.2.1.1. Determine which items should be displayed in the Search Region

You need to review each group and identify items that should not logically be queriable.

If requested to generate search functionality, the JHeadstart Application Generator needs to know what the queriable items are. You can set the **Include in Quick/Advanced Search** properties for each item in a group. Both properties are default true.

1. Select an item in the Application Definition Editor.
2. Check or uncheck the **Include in Quick/Advanced Search** properties.



7.2.2. Using Quick Search

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

1. The item used for searching is always the same. Give the **Quick Search?** property the value singleSearchField. Select the search item in the **Single or Default Search Item** property.
2. You want the user to be able to select the items to search on. Set the **Quick Search?** property to 'dropdownList'. JHeadstart will populate a dropdown list with item names so the user can select the item to query on. Only queriable items are shown in the list (see previous section). The default item is specified by the **Single or Default Search Item** property.

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

7.2.3. 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 several properties that will affect the layout of the Advanced Search Region:

1. The **Form Width** property 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.

Attention: If you use the **Form Width** property when generating a search region for a page of Form layout, this property value will impact the layout of both the search region and the main form 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.
3. **Regions** of the group. If the items you included in the Advanced Search, are also included in a region, then by default a region will also be applied to the advanced search area.

Attention: If you don't want to apply the group regions to the advanced search area, you can use a variation on the template `default\search\advancedSearchRegion.vm`. Comment out the 3 lines just below the comment 'Optional RegionContainer…' by putting `##` in front of each line, and uncomment the 3 lines below the comment 'Use the following code instead…' by removing the `##` in front of each line. Then put those 3 lines instead of the `#ADVANCED_SEARCH_ITEMS()` call within the panelForm above. See the comments in the template, and see the section Using Generator Templates.

7.2.4. Using a Query Operator

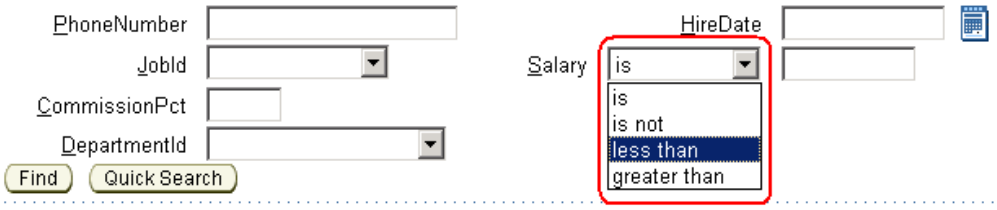
On item level the **Query Operator** can be set. This operator determines how to query the data. Examples are contains, endsWith and greaterThan.

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

You can change this behavior by setting the **Query Operator** property for an item. See the help in the Application Definition 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.

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:



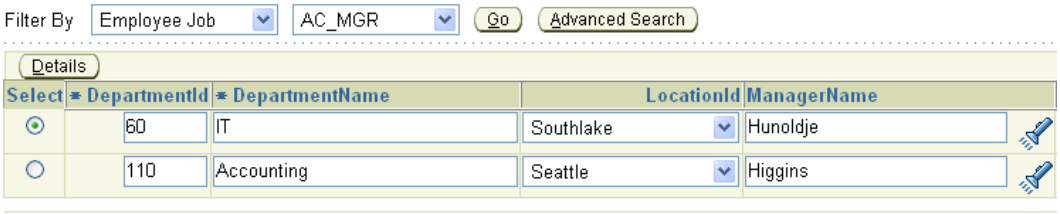
4. JHeadstart has generated a dropdown list with applicable query operators for this field.

7.2.5. Using Query Bind Variables in Quick or Advanced Search

Based on the quick or advanced search items that are set by the user, a query WHERE clause is appended dynamically to the query (see section Search Support in ADF BC Application Module).

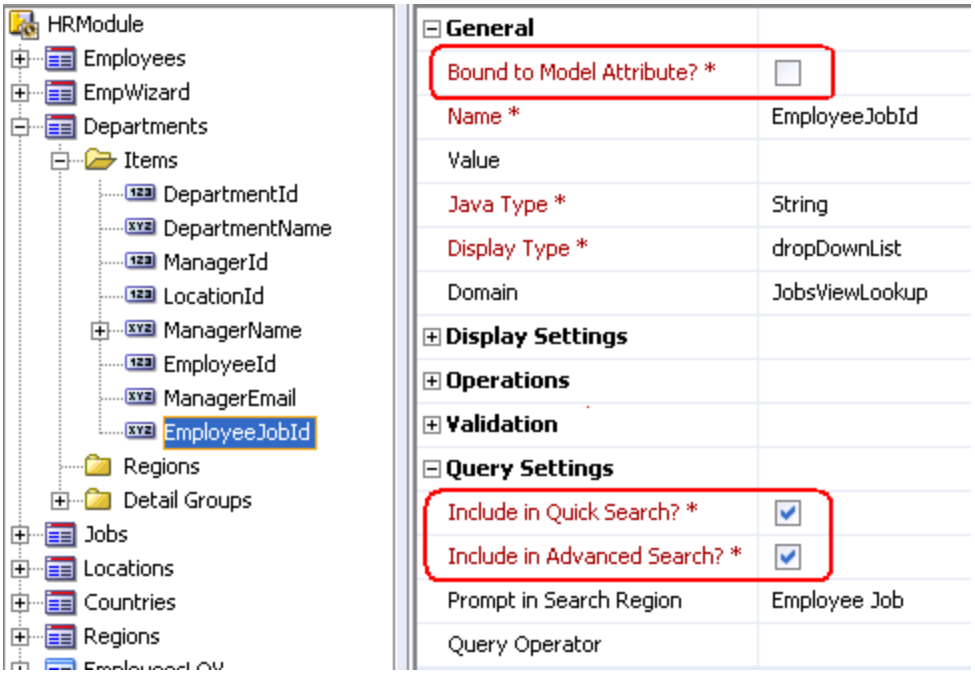
This approach does not allow for adding sub selects to the WHERE clause that references other tables. Since it is a common requirement to perform a search based on values in for example a detail table, JHeadstart allows you to map quick or advanced search items to query bind variables.

Let's use an example to illustrate this technique: we want to be able to search all departments that contain employees with a specific job. In other words, we want to add Employee JobId as a search item to the Departments group.

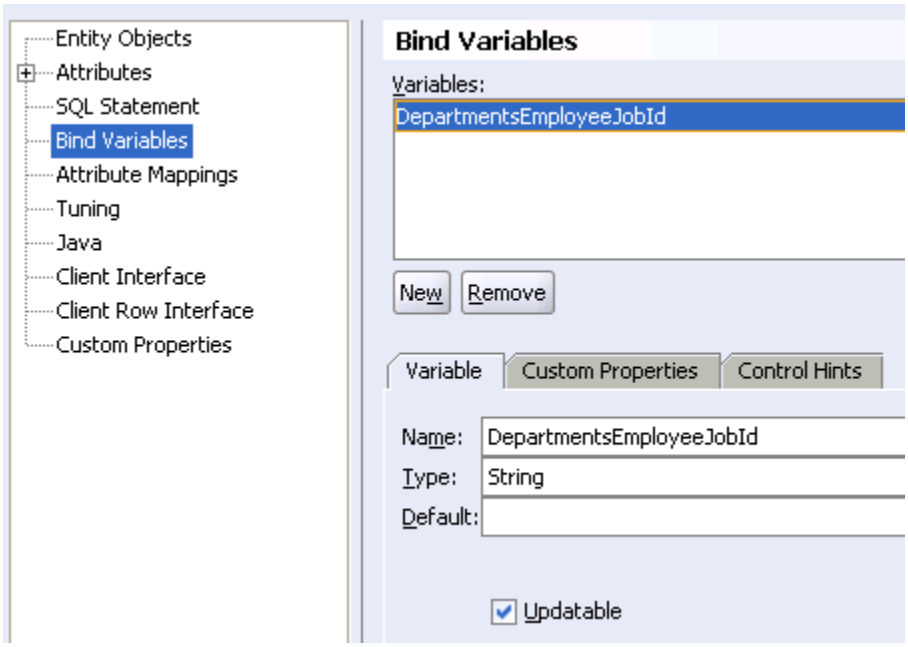


Here are the steps to implement this:

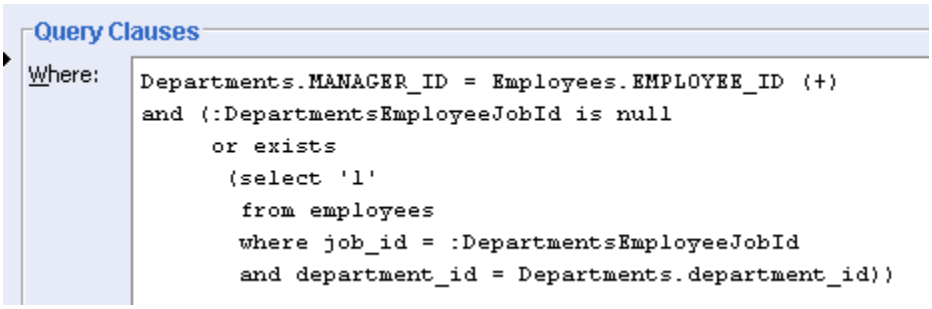
- Add an unbound item "EmployeeJobId" to the Departments group. Do not display this item in form nor table layout, and check the checkboxes "**Show in Advanced Search?**" and "**Show in Quick Search?**".



- In the Departments View Object, defined a named bind variable after the item name, prefixed with the group name: DepartmentsEmployeeJobId.



- In the same Departments View Object add a sub-select to EMPLOYEES table using the value of DepartmentsEmployeeJobId bind variable.



- Generate and run your application!

7.2.6. Runtime Implementation of 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

A search managed bean definition is generated in the group beans faces-config whenever a group has Quick Search and/or Advanced Search enabled. Here is an example of a search bean:

```
<managed-bean>
  <managed-bean-name>searchEmployees</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.bean.JhsSearchBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>bindings</property-name>
    <value>#{data.EmployeesPageDef}</value>
  </managed-property>
  <managed-property>
    <property-name>searchBinding</property-name>
    <value>#{data.EmployeesPageDef.advancedSearchEmployees}</value>
  </managed-property>
  <managed-property>
    <property-name>searchAttribute</property-name>
    <value>EmployeeId</value>
  </managed-property>
  <managed-property>
    <property-name>dataCollection</property-name>
    <value>EmployeesView1</value>
  </managed-property>
  <managed-property>
    <property-name>autoquery</property-name>
    <value>false</value>
  </managed-property>
  <managed-property>
    <property-name>findModeIters</property-name>
    <list-entries>
      <value>#{data.EmployeesPageDef.EmployeesIterator}</value>
    </list-entries>
  </managed-property>
  <managed-property>
    <property-name>maxQueryHits</property-name>
    <value>50</value>
  </managed-property>
</managed-bean>
```

You can see how the generic `JhsSearchBean` class is configured through the managed properties for usage in the Employees page. Related functionality like autoquery and the maximum number of query hits allowed are also implemented through managed properties.

Note the `searchBinding` managed property; this property "injects" the ADF Model action binding that is used to call the `advancedSearch()` method on `JhsApplicationModuleImpl` (see below).

Both the Quick Search Go button and the Advanced Search Find button call a method on the search bean:

```
<af:commandButton action="#{searchEmployees.quickSearch}"
                  textAndAccessKey="#{nls['GO']}"/>

<af:commandButton textAndAccessKey="#{nls['FIND']}"
     action="#{searchEmployees.advancedSearch}" />
```

Internally, the `quickSearch()` and `advancedSearch()` methods delegate the actual work to methods `createArgumentListForAdvancedSearch()` and `executeAdvancedSearchBinding()`.

7.2.6.2. 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 `QueryCondition` 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 page definition of the page (which is of course what the JHeadstart Application Generator does when a Search Region is generated).

The `QueryCondition` 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 `QueryCondition` 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 or source of the `advancedSearch()` method of `JhsApplicationModule` and `JhsApplicationModuleImpl`, and the Javadoc of the `QueryModel` class.

Reference: See the Javadoc or source of the `JhsSearchBean`

7.2.6.3. 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 initially, is governed by the `JhsSearchBean` property `quickSearchMode`. The Quick Search and Advanced Search regions both use an EL expression in the "rendered" property that references the `quickSearchMode` property in the search bean.

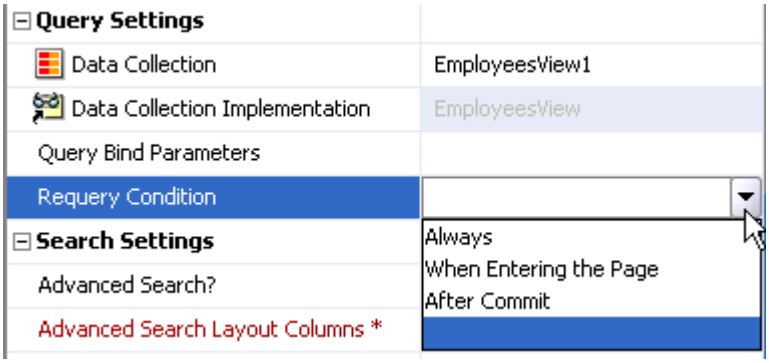
Partial Page rendering is used to switch between Quick Search and Advanced Search. The button 'Advanced Search' looks like this:

```
<af:commandButton id="asButtonEmployees"
                  textAndAccessKey="#{nls['ADVANCED_SEARCH']}"
                  partialSubmit="true"
                  action="#{searchEmployees.switchToAdvancedSearchMode}"/>
```

When this button is pressed, the `switchToAdvancedSearchMode()` is executed which sets the `quickSearchMode` property of the Search Bean 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, since the Search Bean is stored on session scope.

7.3. Forcing a Requery

By default, the ADF Model Iterator Binding created for the data collection of a JHeadstart group is only queried once, and the query results are cached by ADF Business Components. JHeadstart supports the option to force a refresh of the cached query data by using the group level property **Requery Condition**.



This is a combobox property, the drop down list includes three common conditions that JHeadstart translates to a booealn JSF EL expression,an you can also enter a custom Boolean JSF EL expression.

The predefined values are:

- **Always**: every time the user navigates to the page, or submits the page itself, the iterator binding is requeried. This condition translates to the JSF expression **`#{true}`.**
- **When Entering the Page**: the iterator binding is requeried when the user navigates from another page to this page. This condition translates to the JSF expression **`#{!adfFacesContext.postback}`.**
- **After Commit**: the iterator binding is requeried when the user submits the page by clicking on a button that executes the Commit action binding, like the generated Save button. This condition translates to the JSF expression **`#{jhsAfterCommit}`.**

Note that the Requery Condition property is also available for dynamic domains. If you want to refresh the content of a drop down list, you can set the Requery Condition on the dynamic domain.

7.3.1. Implementation of Requery

When the Requery Condition is set, JHeadstart generates two additional entries in the Page Definition of the group:

- An action binding that executes the query.

```
<bindings>
  <action id="ExecuteQueryDepartments" IterBinding="DepartmentsIterator"
          InstanceName="HRModuleDataControl.DepartmentsView1"
          DataControl="HRModuleDataControl" RequiresUpdateModel="true"
          Action="2"/>
```

- An invokeAction in the executables section. The `refreshCondition` of this invokeAction determines whether the executeQuery action binding is invoked or not.

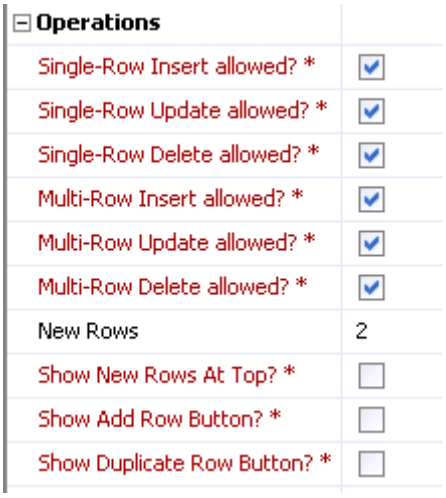
```
<executables>
  <iterator id="DepartmentsIterator" Binds="DepartmentsView1"
            DataControl="HRModuleDataControl" RangeSize="10"/>
  <invokeAction id="ExecuteQueryDepartmentsInvoke"
                Binds="ExecuteQueryDepartments" Refresh="renderModel"
                RefreshCondition="#{jhsAfterCommit}"/>
```

CHAPTER

8

Generating Transactional Behaviors

This chapter describes how you can influence the transactional behavior of generated pages. The properties in the **Operations** group in the Application Definition editor are used for this.



8.1. Enabling Insert

8.1.1. Allowing Inserting Data in a Form Page

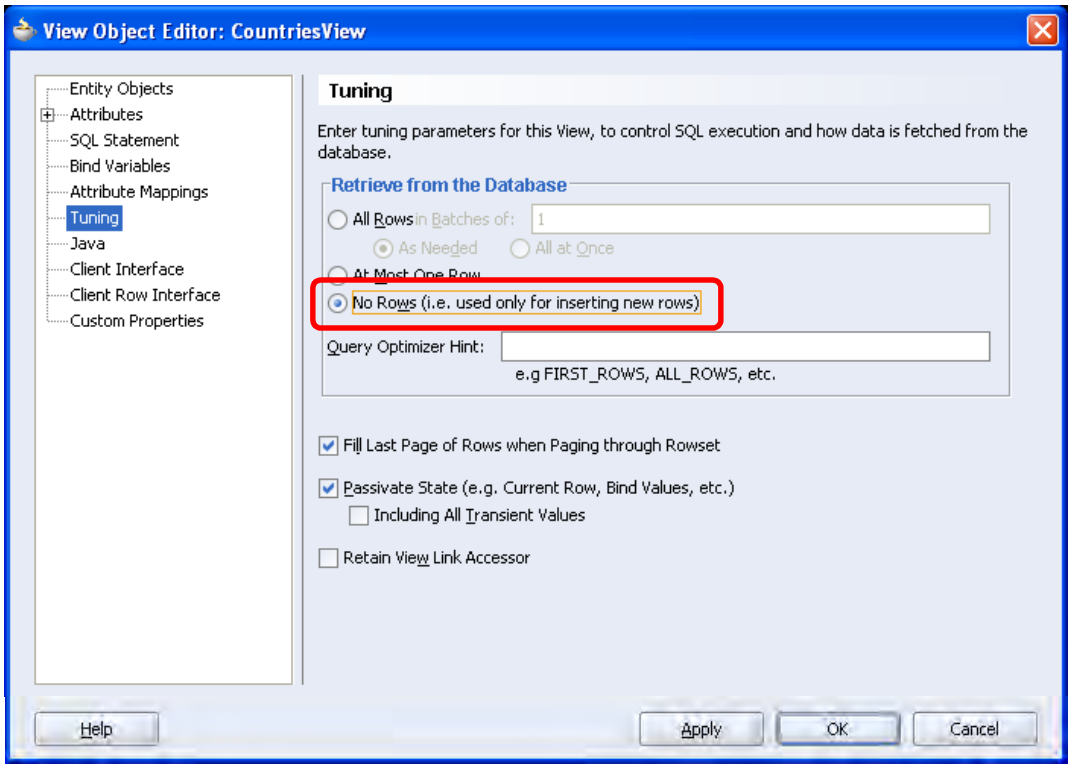
Use property **Single-Row Insert allowed?** JHeadstart will generate a 'New' button on the page.

8.1.2. Building Insert Only Form Pages

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 **Layout Style** to form, **Advanced Search?** and **Quick Search?** properties to none, and disable **Auto Query ?** for the group.

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.



If you test the page by running the page directly from JDeveloper, you will see that it does not start with a new empty row. This is because with insert-only pages, the creation of a new row is performed when JSF navigates to a special Start[groupName] outcome, which occurs if you navigate to the page from a menu 1 tab for example. So always test the page through clicking a tab, or by creating a home page that uses the Start[groupName] outcome. If you nevertheless want to start the page directly in insert mode, you should include a request parameter Start[groupName]=true in the URL that you use to start the page.

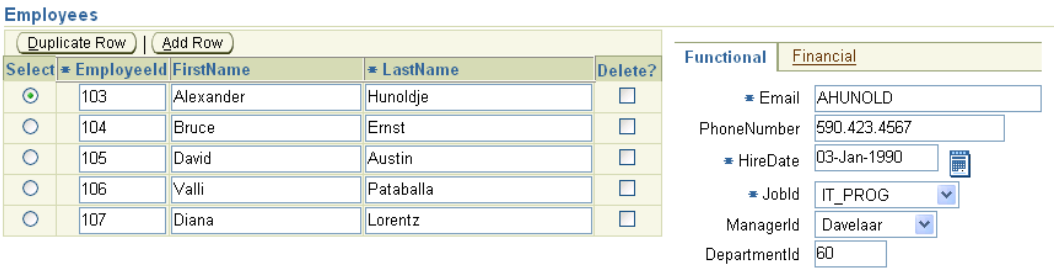
8.1.3. Allowing the User to Insert Data in a Table Page

Inserting rows in a table layout is supported in three ways:

- By displaying additional empty rows in the table using the **New Rows** property. The **Show New Rows at Top** property determines whether the new rows are displayed as the first or last rows in the table. Note that when **Show New Rows at Top** is unchecked, and you use table ranges then the empty rows are only visible when the user navigates to the last range set of rows.
- By checking the **ShowAdd Row Button** checkbox.
- By checking the **Show Duplicate Row Button** checkbox.

All of the above properties are only applicable when the **Multi-Row Insert Allowed** checkbox is checked.

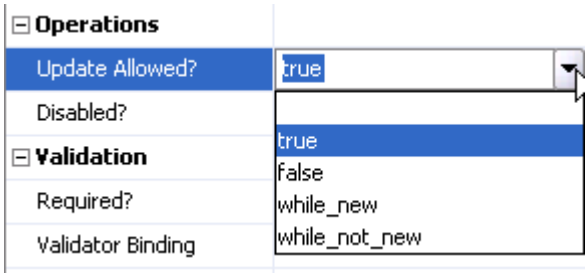
You can even use all three of them together. However, the **New Rows** property cannot be set when using **Table Overflow Style** "below" or "right".



8.2. Enabling Update

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

Which items will be updateable or not depends on the **Update Allowed** property of an item. This property can be used to make items read only, always updateable, or updateable in a new row, or updateable in an existing row.



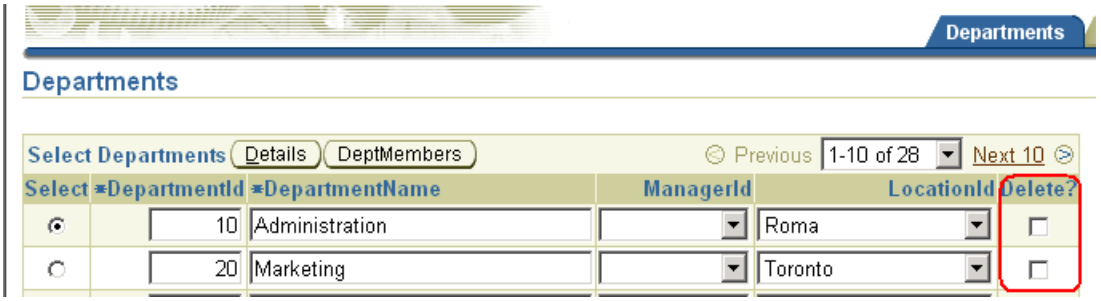
The **Update Allowed** property is a combo box, you can choose one of the options from the drop down list, and you can also enter a boolean JSF EL Expression.

8.3. Enabling Delete

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

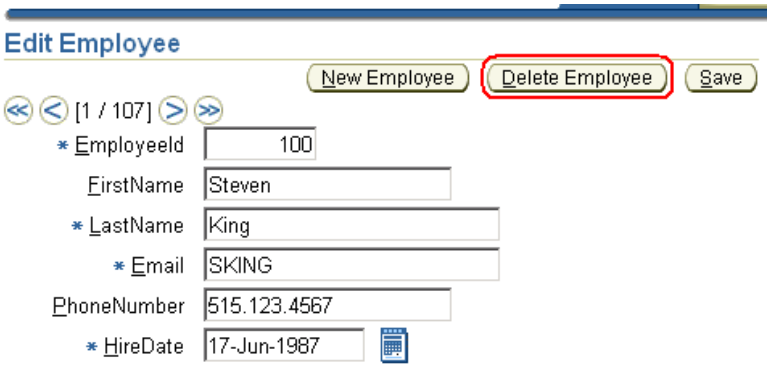
The JHeadstart Application Generator generates a Delete button on a single row page, or a delete check box on a multi row page.

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':



8.4. Conditionally Enabling Insert, Update an Delete

The previous sections explained how to enable or disable insert, update and delete functionally for the group, regardless of user roles and permissions or the actual data.

To conditionally enable insert, update and delete operations you can use the **Insert Allowed EL Expression**, **Update Allowed EL Expression** and **Delete Allowed EL Expression**. These properties are available both on the Service-level and on Group level. On service-level, you will typically use them in combination with permission-based access control. See chapter 10 "Application Security", section 10.7 "Restricting Group And Item Operations based on Authorization Information" for more information.

On the group level you can refer to user roles, and you can make the operation enabled based on the actual data shown on the page.

For example, if you have a business rule which specifies that users in the HR_ASSSISTANT role can only delete job PU_CLERK and that HR_MANAGERS can delete all jobs, then you can specify the following **DeleteAllowed EL Expression** to implement this rule:

```
#{jhsUserRoles['HR_MANAGERS'] or (jhsUserRoles['HR_ASSISTANT'] and
bindings.JobsJobId.inputValue=='PU_CLERK')}
```

users can only delete jobs where the Delete button in the Edit Jobs page should only be available when the Job Id is

8.5. Runtime Implementation of Transactional Behaviors

8.5.1. Multi-Row Insert and Delete

Out-of-the-box, JSF only supports editing of existing rows in a table. The Update Model phase does not support multi-row insert and delete.

To implement multi-row insert and multi-row delete in a table, JHeadstart uses the class `oracle.jheadstart.controller.jsf.bean.JhsCollectionModel`. This class extends the default ADF Faces collection model class. The value property of an ADF Faces table references a managed bean that uses the `JhsCollectionModel` class, like this:

```
<af:table value="#{Departments2CollectionModel}" ... />
```

The corresponding managed bean definition looks like this:

```
<managed-bean>
  <managed-bean-name>Departments2CollectionModel</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.bean.JhsCollectionModel
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>jhsPageLifecycle</property-name>
    <value>#{jhsPageLifecycle}</value>
  </managed-property>
  <managed-property>
    <property-name>bindings</property-name>
    <value>#{bindings}</value>
  </managed-property>
  <managed-property>
    <property-name>rangeBinding</property-name>
    <value>#{bindings.Departments2Table}</value>
  </managed-property>
  <managed-property>
    <property-name>newRowCount</property-name>
    <value>2</value>
  </managed-property>
  <managed-property>
    <property-name>newRowsAtTop</property-name>
    <value>false</value>
  </managed-property>
  <managed-property>
    <property-name>newRowList</property-name>
    <list-entries>
      <value-class>oracle.jheadstart.controller.jsf.bean.NewTableRowBean
      </value-class>
      <value>#{NewDepartments2TableRow1}</value>
      <value>#{NewDepartments2TableRow2}</value>
    </list-entries>
  </managed-property>
</managed-bean>
```

As you can see, the generic `JhsCollectionModel` class is configured for specific usage in the DepartmentsTable using managed property settings. The managed property `newRowCount` determines the number of empty rows that will be displayed in the table that can be used to insert new rows.

Each `JhsCollectionModel` instance "registers" itself as a "Model Updater" in `JhsPageLifecycle`. Just before model validation takes place, the `JhsPageLifecycle` class calls the `doModelUpdate()` method in each of the registered Model Updaters. The `doModelUpdate()` method in turn calls method `processNewRows()` to insert any new rows when the user entered data in the new lines of the table. Method getRowsToRemove() is called to determine the list of rows that must be deleted, based on whether the delete checkbox is checked for a specific row in the table.

Before actually performing the deletes, 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 row was still present in the database.

Reference: See the Javadoc or source of `JhsCollectionModel` and `JhsPageLifecycle`.

8.5.2. Single-Row Insert

A 'Create' action binding is executed when a 'New …' button is pressed. The `JhsPageLifecycle` class has overridden method `invokeActionBinding()` to intercept the execution of any Create action binding, and call the `onCreate()` method in the same class.

The `onCreate()` method first creates the new row, then tries to lookup a default values managed bean with the same name as the Create action binding, suffixed with "DefaultValues". When such a bean is found, the `applyDefaultValues()` method is called on this bean.

Finally, it populates the `createModes` managed bean Hashmap with an entry with the create action binding name as the key and `Boolean.TRUE` as the value. This entry causes the page to be rendered in insert mode. See also chapter 5 "Generating Page Layouts" section "Create/Update Mode"..

8.5.3. Single-Row Delete

A 'Delete' action binding is executed when a 'Delete …' button is pressed. The `JhsPageLifecycle` class has overridden method `invokeActionBinding()` to intercept the execution of any Delete action binding, and call the `onDelete()` method in the same class.

The `onDelete()` method first passivates the state of the application module, then it deletes the row, and then it calls the `onCommit()` method to remove the row from the database. The passivation of Application Module state is needed so we can restore the deleted row in ADF BC when the actual database commit fails.

Reference: See the Javadoc or source of `JhsPageLifecycle.onDelete()`.

8.5.4. Commit Handling

The 'Commit' action binding is executed when the 'Save' button is pressed. When using JSF, the ADF Model layer uses a Faces specific class (`FacesCtrlActionBinding`) at runtime to execute the action binding. This class wraps the actual action binding class (`JUCtrlActionBinding`) and checks the active Page Lifecycle class for a method named `onCommit()`. If such a method is found, it calls this method. If such a method is not found, it simply calls the `execute()` method on the wrapped action binding class.

The `JhsPageLifecycle` class includes such an `onCommit()` method to do the following:

- When the user tries to save data without having made any changes, he gets a 'No changes to save message' (JHS-00101). JHeadstart checks the state of the application module to determine whether there are outstanding changes. The message is added to the JSF message stack.
- When the commit succeeds, a 'Transaction completed' message (JHS-00100) is added to the JSF message stack.
- When the commit fails, the passivated state of the application module is activated again to bring back any rows that might already have been removed from the corresponding ViewObject iterator, when the Commit failed because of a database error.

If you do not want to display the JHS-0010 or JHS-00101 messages, or you want to replace them with a different message (either in all pages, or in a specific page) you can create your own Page Lifecycle managed bean and set the managed properties `transactionCompletedMessageKey` and/or `noChangesMessageKey` to the desired values. See chapter 11 ADF-JSF page Lifecycle for more information on how to do this.

8.5.4.1. Add Commit Behavior to a Custom Button

If you want to perform the same Commit handling on a custom button that invokes a business method (see section 6.11 "Custom Button that Calls a Custom Business Method"), you can do so as follows:

- In the page visual design editor, double-click on the button. This will open a dialog where you can create a managed bean method that is called when the button is pressed, and ADF will prepopulate the button method with the code required to execute the action binding that calls the business method.
- After the code that executes the action binding, you can add the following code to fire the onCommit logic of the JhsPageLifecycle:

```
JhsPageLifecycle lifecycle = JhsPageLifecycle.getInstance();
FacesPageLifecycleContext context =
    (FacesPageLifecycleContext) lifecycle.getLifecycleContext();
lifecycle.onCommit(context);
```

If you want to override the JHS-00100 and JHS-00101 message for this specific button, you can use the overloaded version of onCommit that accepts the resource keys for transaction-completed message, and no changes to save message:

```
- JhsPageLifecycle lifecycle = JhsPageLifecycle.getInstance();
  FacesPageLifecycleContext context =
      (FacesPageLifecycleContext) lifecycle.getLifecycleContext();
  lifecycle.onCommit(context,"ACM-00003","ACM-00004");
```

Reference: See the Javadoc or source of `JhsPageLifecycle.onCommit()`.

8.5.5. 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 takes care of rolling them back.

Looking at a generated commandMenuItem will clarify how this works:

```
<af:commandMenuItem text="Employees" onclick="return alertForChanges();"
                    action="StartEmployees" immediate="true"
                    selected="#{attrs.selectedTab=='Employees'}">
  <f:actionListener type="oracle.jheadstart.controller.jsf.listener.DoRollbackActionListener"/>
  <f:actionListener type="oracle.jheadstart.controller.jsf.listener.ResetBreadcrumbStackActionListener"/>
  <af:resetActionListener/>
</af:commandMenuItem>
```

When the user navigates out of a page with pending changes, he will get a JavaScript warning as is explained in section 11.5 Outstanding Changes Warning.

When the user still wants to leave the page, the action listeners defined against the commandMenuItem will fire. The first action listener calls the JHeadstart `DoRollbackActionListener` class which lookups the Rollback action binding and executes it. This fires the `onRollback()` method in `JhsPageLifecycle` in the same way as the `onCommit()` is fired when executing the Commit binding (see the previous section). The `onRollback()` method stores the keys of the current row of each View Object in the current binding container. After that, the actual rollback is executed and the current rows are reset to the values before the rollback.

Reference: See the Javadoc or source of `JhsPageLifecycle.onRollback()`, `JhsPageLifecycle.storeRowCurrencies()`, and `JhsPageLifecycle.restoreRowCurrencies()`.

CHAPTER

9

Creating Menu Structures

JHeadstart supports two styles of creating menus:

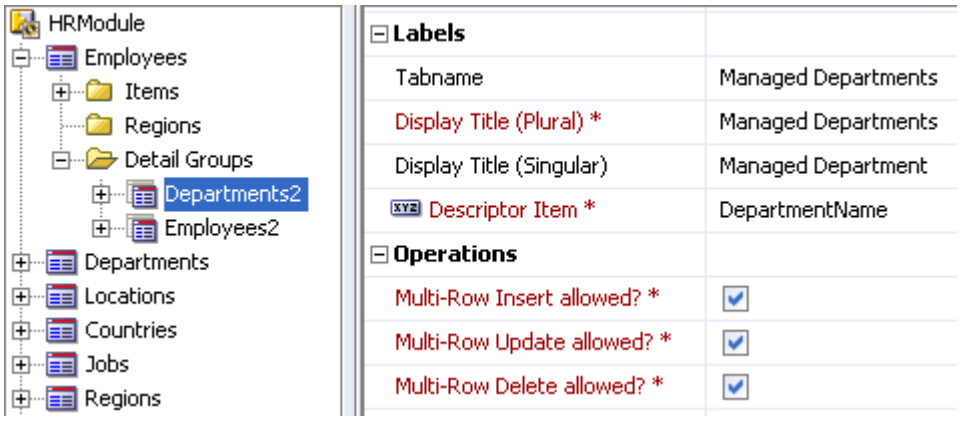
- a static menu structure generated based on the group structure of the application definition
- a dynamic menu structure which is table-driven, and can be configured at runtime using menu administration screens

Both menu styles define the *structure* of the menu; (custom) menu templates will determine the actual *layout* of the menu. In this chapter, we will first explain the two ways of creating a menu structure; the last section discusses how you can change the layout of the menu.

9.1. Static Menu Structure

By default, JHeadstart generates a static menu structure that reflects the structure of the service as defined in the application definition. For each top-level group, a tab within the level 1 menu tab bar is generated. For second-level groups that are not displayed on the same page as the top-level group, a tab within the level 2 menu tab bar is generated, as shown in the pictures below.

Group structure defined in application definition



determines generated menu 1 and menu 2 tab bars

The **Tabname** property of a group determines the label of the menu tab. When you check the service-level checkbox **Generate NLS-enabled prompts and tabs?,** the label of the menu option will be read from a resource bundle, with the value of the **Tabname** property used as default. See the section on Internationalization for more info on multi-language support.

9.1.1. Which Menu Tab is Selected?

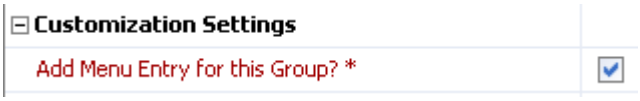
When using the static menu structure, the level 1 and level 2 menu tabs that are displayed as selected are determined by the current page. Each generated page contains a page snippet that references the ADF Faces Region .jspx file that contains the menu1 and menu 2 entries. In this reference, the selected tab name is passed as shown in the page snippet below:

```
<!-- DEBUG:BEGIN:PAGE_MENU : default/misc/menu/pageMenu.vm, nesting level: 2 -->
<f:facet name="menu1">
  <af:region id="HRModuleMenulTabs" value="#{bindings}"
      regionType="view.region.HRModuleMenulTabs">
  <af:attribute name="selectedTab" value="Employees"/>
    </af:region>
</f:facet>

<f:facet name="menu2">
  <af:region id="HRModuleEmployeesMenu2Tabs" value="#{bindings}"
      regionType="view.region.HRModuleEmployeesMenu2Tabs">
  <af:attribute name="selectedTab" value="Employees"/>
    </af:region>
</f:facet>
<!-- DEBUG:END:PAGE_MENU : default/misc/menu/pageMenu.vm, nesting level: 2-->
```

To customize the selected menu tab, you need to customize the pageMenu.vm template, as discussed in the section "Customizing the Static Menu Structure".

9.1.2. Preventing Generation of a Menu Tab

If you do not want a menu entry to be generated for a given group, you can uncheck the group-level property **Generate Menu Entry for this group**.



Note that this property is only visible when you have switched the Application Definition Editor to expert mode

9.1.3. Customizing the Static Menu Structure

The menu level 1 entries and menu level 2 entries are not hardcoded into each and every page. Instead they are generated into a separate ADF Faces Region files. The menu level 1 entries are generated using the MENU1 template default/misc/menu/menu1.vm and the level 2 entries are generated using the MENU2 template default/misc/menu/menu2.vm.

The reference to both ADF Faces region files in each generated page is handled by the PAGE_MENU template default/misc/menu/pageMenu.vm. So, if you want to restructure the generated menu structure, you can create custom templates for these three templates.

Let's use a simple example to illustrate this technique. Instead of the default row of menu1 tabs, we want the menu 1 entries to be displayed vertically at the left side of the page, as shown in the screen shot included below.

To achieve this, we must place the menu1 inside the menu3 facet of the <af:panelPage> element. Here are the steps to do this:

- Create a custom template for pageMenu.vm, and rename the menu1 facet to menu3:

```
<f:facet name="menu3">
  #set ($includeName = "${JHS.service.name}MenulTabs")
  #INCLUDE_OPEN($includeName ${JHS.menuGenerator.menul})
   #INCLUDE_ATTRIBUTE($includeName "selectedTab" ${JHS.cu
  #INCLUDE_CLOSE($includeName)
</f:facet>
```

- Create a custom template for menu1.vm, and rename the enclosing element from <af:menuTabs> to <af:menuList>

```
<af:menuList>
#foreach ($menuItem in $JHS.current.menu.menuItems)
  <af:commandMenuItem text="${JHS.nls($menuItem.page.
                      action="${JHS.facesConfigGenera
                      selected="#{attrs.selectedTab==
                      #if ($menuItem.page.group.roles
                        rendered="#{jhsUserRoles['$me
                      #end
                      >
    <f:actionListener type="oracle.jheadstart.control
    <f:actionListener type="oracle.jheadstart.control
    <af:resetActionListener/>
  </af:commandMenuItem>
#end
</af:menuList>
```

Now configure JHeadstart to use your custom menu templates instead of the default templates. See chapter 4 “Using JHeadstart”, section “Customizing Using Generator Templates” for more information on how to do this. That’s all, regenerate your application and the menu1 will look like in the above screen shot.

Note that the menu3 facet is also used to render a tree in case of a tree-form layout. So, this menu layout cannot be used in combination with tree layouts.

Another common scenario is to have the generated menu1 entries displayed as menu2 entries, to be able to add a top-level menu that provides access to the various subsystems. In this scenario the "old" menu2 level entries are no longer displayed. This is possible because JHeadstart also generates buttons to navigate to the detail groups that are made accessible through the default level 2 menu.

Here are the steps to do this:

- Create a copy of the menu1.vm template, and replace the forEach loop with hardcoded top level menu entries you want to appear. For example:

```
<af:menuTabs>
  <af:commandMenuItem text="${JHS.nls("Human Resources","HUMAN_RESOURCES_MENU_LABEL")}"
                      #ALERT_FOR_CHANGES()
                      action="StartEmployees" immediate="true"
                      selected="true"
                      >
    <f:actionListener type="oracle.jheadstart.controller.jsf.listener.DoRollbackActionListener"/>
    <f:actionListener type="oracle.jheadstart.controller.jsf.listener.ResetBreadcrumbStackActionListener"/>
    <af:resetActionListener/>
  </af:commandMenuItem>
  <af:commandMenuItem text="${JHS.nls("Sales","SALES_MENU_LABEL")}"
                      #ALERT_FOR_CHANGES()
                      action="StartEmployees" immediate="true"
                      selected="false"
                      >
    <f:actionListener type="oracle.jheadstart.controller.jsf.listener.DoRollbackActionListener"/>
    <f:actionListener type="oracle.jheadstart.controller.jsf.listener.ResetBreadcrumbStackActionListener"/>
    <af:resetActionListener/>
  </af:commandMenuItem>
</af:menuTabs>
```

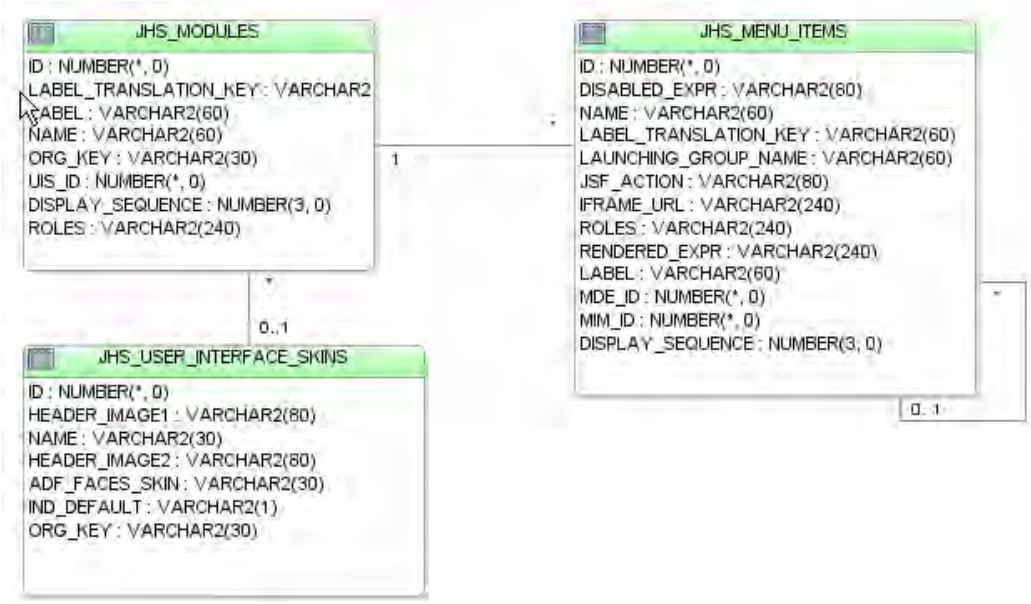
- Create a custom template for pageMenu.vm, rename menu1 facet to menu2, and change the includeName suffix to menu2Tabs.
- Remove the existing menu2 facet (and surrounding if statement) from the template.
- Add a new menu1 facet that uses the JHS_PARSE_INCLUDE macro to create an ADF Faces region file with the content of your copied menu1.vm (myTipMenu.vm) template. See the code snippet below for an example, and note the reference to the actual template path and name.

```
<f:facet name="menu1">
  #JHS_PARSE_INCLUDE("${JHS.service.name}MenuTabs" "hr/misc/menu/myTopMenu.vm" ${JHS.current.model})
</f:facet>

<f:facet name="menu2">
  #set ($includeName = "${JHS.service.name}Menu2Tabs")
  #INCLUDE_OPEN($includeName ${JHS.menuGenerator.menu1})
   #INCLUDE_ATTRIBUTE($includeName "selectedTab" ${JHS.current.group.baseGroup.name} "java.lang.String" "false")
  #INCLUDE_CLOSE($includeName)
</f:facet>
```

- Create a custom template for menu1.vm, and rename the enslosing element <af:menuTabs> to <af:menuBar>
- Configure JHeadstart to use your customized menu1.vm and pageMenu.vm templates and regenerate your application.

9.2. Dynamic Menu Structure

Although the static menu structure can be changed using custom templates, it quickly becomes a tedious job to do so when you have a large application and the required menu structure is rather different from your group structure. In such situations, it is easier and more flexible to use a dynamic menu structure that can be configured and customized at runtime by a system administrator. JHeadstart uses a set of database tables to support these dynamic menus. The structure of these tables is shown below.



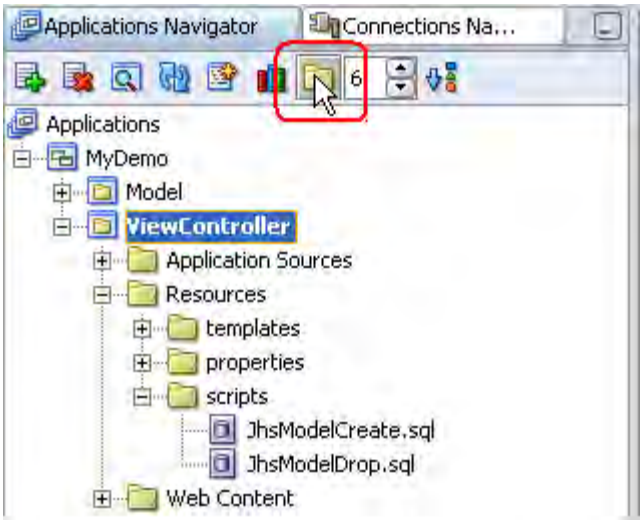
The top level of the menu structure is defined by so-called modules. A module can be seen as a logical subsystem of your application, and will often match with a service defined in one application definition, but this is not required. Each module has a nested structure of menu items. A menu item can simply launch the first page of a group as defined in an application definition file, but you can also specify a custom JSF Navigation action, or some arbitrary URL that will be displayed inside an iFrame within the page.

You can also associate a user interface skin with a module; this allows you to support multiple user interface skins depending on the currently selected module.

In the next sections we will explain how you can enable your application to use a dynamic menu structure.

9.2.1. Creating the Database Tables

Before you can start using dynamic menus, you need to create the above table structure in your own application database schema. You can do this by running the script JhsModelCreate.sql against the database connection of your application schema. This script is located in the scripts directory of your ViewController project. If you don't see the scripts directory, make sure you click the Toggle Directories in the toolbar of the Application Navigator.



You can right-mouse-click on the JhsModelCreate.sql, then choose Run in SQL*Plus, and then the database connection you want to run the script in.

Attention: We recommend installing the JHeadstart tables in the same schema as your own application tables. If you nevertheless prefer to install the JHeadstart tables in a different database schema, then you need to ensure that your application schema has full access to the JHeadstart tables and synonyms with the same name as the table name. This is required because the JHeadstart runtime accesses the database tables through View Object usages defined in application module **JhsModelService**. When generating your application while using one or more of the table-driven features, this JhsModelService application module is added as a nested usage to your own application module, thereby "inheriting" the database connection of its parent application module.

Attention: The JhsModelCreate.sql script creates database tables for all table-driven JHeadstart runtime features. Additional tables for flex items, translations and security are also created. If you do not plan to use these other features you can create your own script that only creates the above tables, and the JHS_SEQ sequence that is used to populate the ID column in these tables.

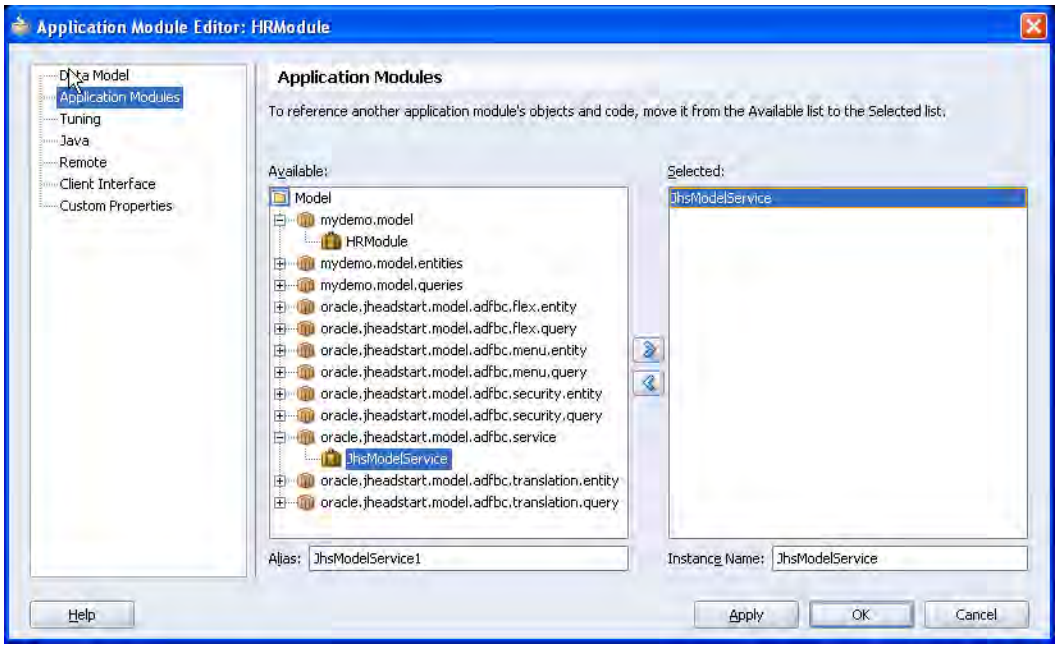
9.2.2. Enabling Dynamic Menus

To enable your application for use of dynamic menus, the first thing to do, is to check the service-level checkbox **Allow Runtime Customization of Menu?.**

9.2.2.1. Running the JHeadstart Application Generator

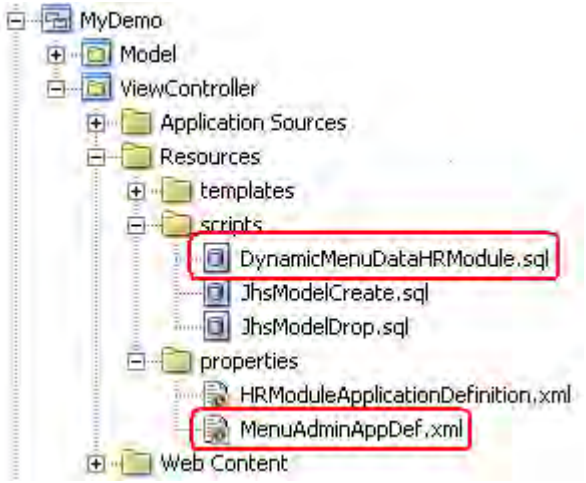
When you now run the JHeadstart Application Generator again, the following happens to enable dynamic menus:

- All ADF Business Components included in the JHeadstart Runtime library are imported into your Model Project, and the **JhsModelService** application module, is added as a nested usage to your own application module. The **JhsModelService** includes View Object Usages that insert, update, delete and query the underlying database tables needed for the dynamic menu. Note that by creating JhsModelService as a nested application module, it will inherit the database connection of the parent application module.



- An additional Application Definition file, named MenuAdminAppDef.xml is generated. Click Save All after running the JAG and this file should appear into the properties directory of your ViewController project. You can use this Application Definition to generate the pages that are used to define the menu structureat runtime. See section Generating the Menu Admin Pages for more info.

- A SQL Script named DynamicMenuData*ServiceName*.sql is generated and executed against the default database connection of your ADF Business Components project. This script inserts one row in the JHS_MODULES table for the service, and multiple rows in the JHS_MENU_ITEMS table for each top-level group and second-level group defined in the Application Definition. This SQL script is just created to provide you with a default menu structure that you can use to test your application. Note that if you do not want the JAG to auto-execute the script, you can uncheck the **service-level** checkbox "**Run Generated SQL Scripts?"**



- A module-switcher drop-down list is generated in the global buttons area of the page. This drop-down list allows you to switch modules in your application, which causes a different menu structure to be displayed as well: the menu items defined for the selected module.

9.2.2.2. Generating the Menu Admin Pages

As explained above, a separate application definition MenuAdminAppDef.xml is generated that can be used to generate the menu administration screens. This application definition is only generated when it does not exist yet, so after it has been generated, you can make any changes you want using the Application Definition Editor, without loosing these changes when you regenerate your "own" application definition.

Before you generate the MenuAdminAppDef you might want to inspect the file locations properties, and change them if you have set other naming standards for these properties.

The default settings are in line with what we recommend:

- Generate all Main faces-config files in the WEB-INF directory with a suffix indicating the service
- JhsCommon-beans.xml is shared by all services (application definitions); the location should never be changed.
- Create a subdirectory per service for the generation of group beans faces config files, pages and regions. Make a subdirectory under this "service" directory for each of these file types.
- The **View Package** property should be the same across all services
- The **Resource Bundle Type** and **NLS Resource Bundle** properties should be the same across all services.

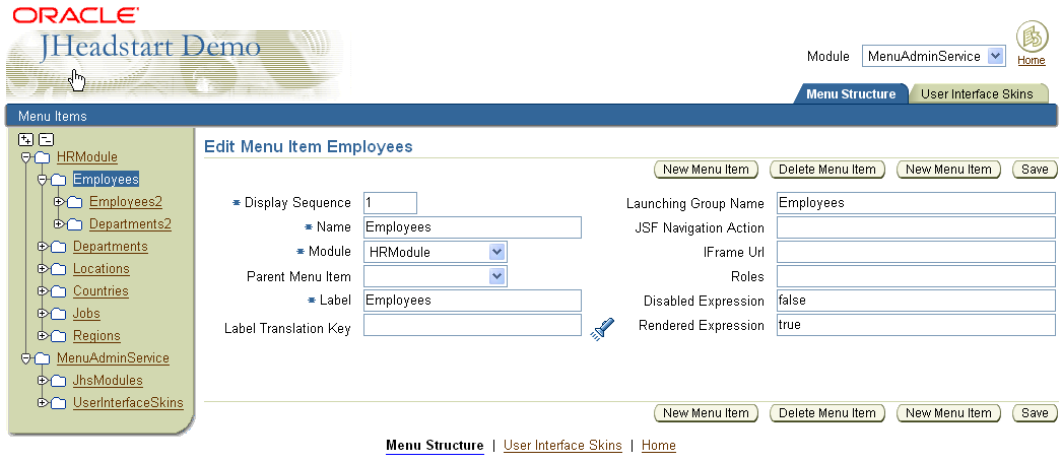
Note that all service-level properties that are generally the same across all services have got the same values as in your "own" application definition. If you later on decide to make changes to these settings in your own application definition, then you will need to make the same change in the MenuAdminAppDef Application Definition.

Now, if you are satisfied with the settings, you can run the JAG for the MenuAdminAppDef. Note that like every other Application Definition service, the Menu Admin service is treated as another module in your application: the generated SQL script DynamicMenuDataMenuAdminService.sql has inserted a row in JHS_MODULES, and rows in JHS_MENU_ITEMS for all level 1 and level 2 groups.

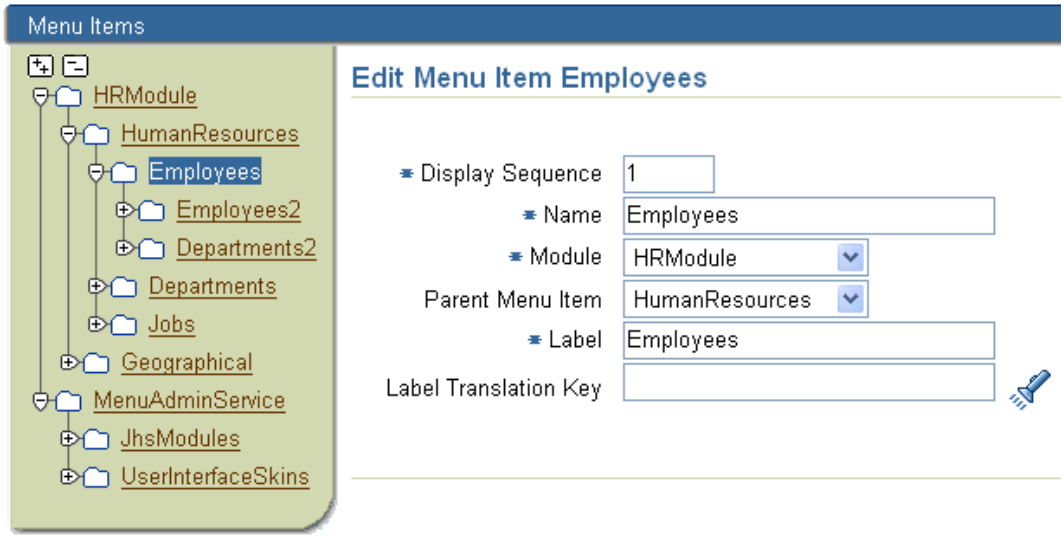
9.2.3. Defining the Menu Structure At Runtime

You are now ready to run the generated application, and change/define the menu structure at runtime. If you start the application again, you will notice that the menu structure is quite similar to the static menu structure.

This is because the generated SQL script that inserts rows in the JHS_MODULES and JHS_MENU_ITEMS tables generates entries using the same algorithm as used for the static menu structure. Now, using the module drop down list, we can navigate to the MenuAdmin module, and change the menu structure anyway we want.



For example, within the HRModule, lets create only two top-level menu entries “Human Resources” with Employees, Departments and Jobs as level-two menu entries, and “Geographical”, with Regions, Countries and Locations underneath. We can do this by creating new menu items directly under the HRModule, and making the appropriate new menu item the parent menu item of the existing first-level menu items, effectively changing them into level-two menu entries. The new structure will look like this:



Now, if you navigate back to the HRModule using the module drop down list, the menu will look like this:

Note that “Employees2” and “Departments2” are now turned into level three menu items and are no longer visible in the menu structure. This is because the default menu layout templates only render level one and level two menu items. You can change this using custom templates as explained in section “Customizing Menu Layout”.

9.2.4. Linking a User Interface Skin to a Module

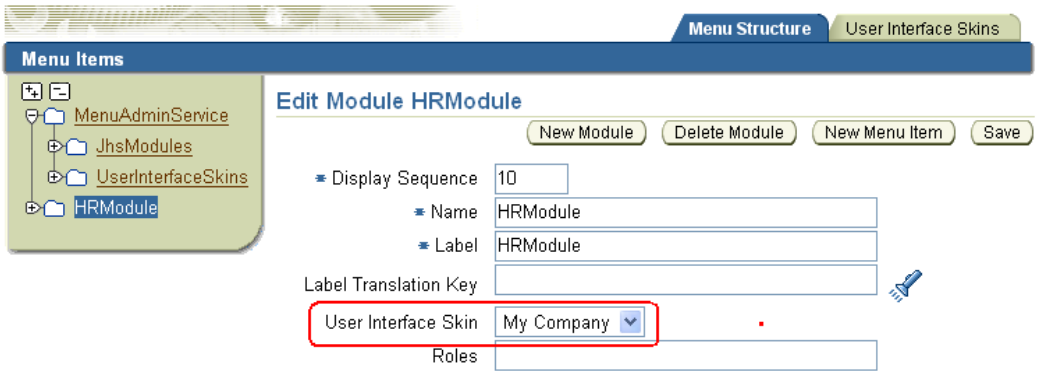
If you have the requirement to render each module within your application with a different look and feel, then you can accomplish this by defining so-called User Interface Skins, and link such a skin to a module. You can define a skin through the Menu Administration service.

A JHeadstart User Interface Skin integrates with the ADF Faces Skinning feature as shown in the above screen shot. The value of the **ADF Faces Skin** property must be a value that exists in the adf-faces-skins.xml file, located in web.xml.

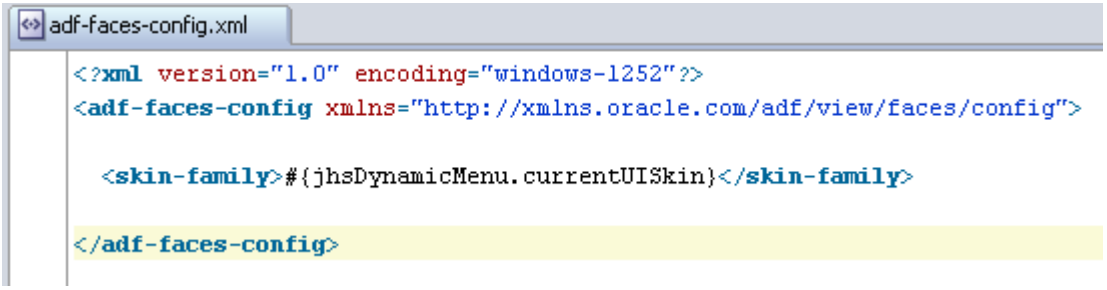
Web Link: More information about using ADF Faces Skinning and creating your own skin can be found in section 22.3 Using Skins to Change the Look and Feel of the ADF Developers Guide.

The mycompany skin used as an example in this section, can be downloaded from OTN as well.

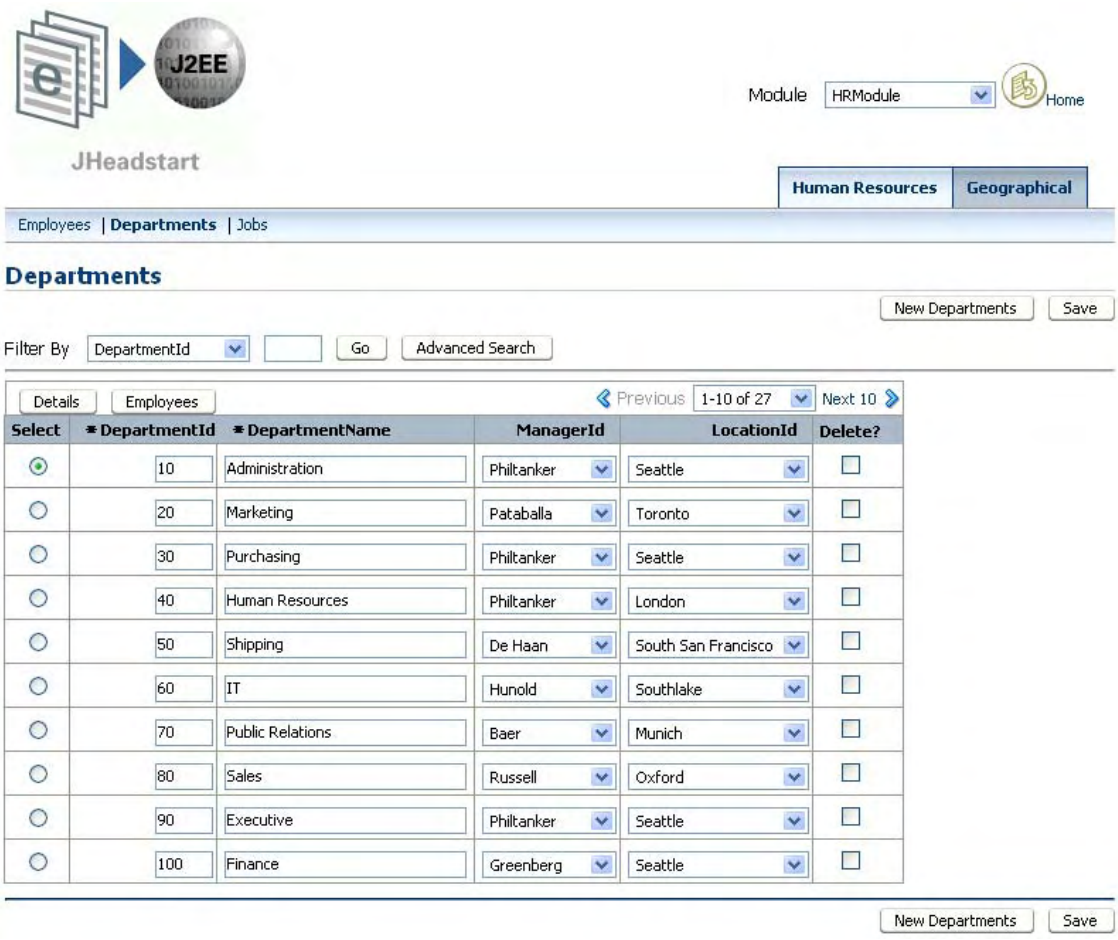
If you have defined one or more user interface skins, you can associate a skin with a module.



Now, to enable this dynamic skin switching based on the currently selected module, you need to make a change in adf-faces-config.xml, to configure ADF Faces to read the skin to use from the *jhsDynamicMenu.currentUISkin* managed bean property:



If you now restart the application, and select the HRModule, the *mycompany* skin is applied, which will look like this:



Note that the header image has changed as well, it is now set to the JHeadstart logo as specified in the User Interface Skin maintenance page.

Modules that do not have a skin specified continue to be rendered with the default oracle skin.

This page is intentionally left blank.

CHAPTER

10 Application Security

Application security in ADF web applications can be implemented at many levels:

- In the browser by using the secure https protocol, and browser certificates.
- In the View and Controller tiers by restricting access to web pages, by hiding UI controls that provide access to unauthorized application functions, and/or by making UI Input controls updateable or read only based on user roles.
- In the ADF Model tier by using ADF Security to restrict access to page definitions, and to executables and bindings within the page definition.
- In the Business Service tier by restricting read, insert and update access to ADF Business Components.
- In the Database tier by restricting access to specific data objects, for example by granting select, insert, update and delete privileges to users and user roles. When all your application users connect to the database using the same database user (the rule rather than the exception in browser-based applications), you can use Oracle Row Level Security (RLS) and Oracle Virtual Private Database (VPD) to implement access privileges.

This chapter focuses on the extensive support provided by JHeadstart and ADF to implement security in the View, Controller, Model and Business Service tier.

10.1. Understanding and Choosing Security Options with JHeadstart

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 Java EE web applications:

- Container-managed security using JAAS (Java Authentication and Authorization Service)
- Custom security

JHeadstart supports both types of security, and even allows you to combine both approaches by using custom roles and/or permissions in addition to JAAS-based security. This paragraph discusses the security options in more detail, which should help you in configuring the JHeadstart security settings in your Application Definition (at Service level).

Security	
Authentication Type	JAAS
Use Role-based Authorization? *	☑
Authorization Type	JAAS
Authorize Using Group Permissions? *	☑
Role/Permission Prefix	
Administrator Role	ADMIN
User Role	USER
Insert Allowed EL Expression	#{jhsUserRoles['$GROUP_NAME$.create']}
Update Allowed EL Expression	#{jhsUserRoles['$GROUP_NAME$.update']}
Delete Allowed EL Expression	#{jhsUserRoles['$GROUP_NAME$.delete']}
When Access Denied go to Next Group? *	☑

10.1.1. JAAS and JAZN

Within the JEE platform, Java Authentication and Authorization Services (JAAS) is the standard for implementing security. By using JAAS the web container used to run the web application enforces proper authentication and authorization ("container-managed security"). Each web container provides its own implementation of the JAAS standard.

JAAS allows web developers to develop the security in their application 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'".

To get the name of the currently logged in user in a JSF application, the following statement can be used:

```
FacesContext.getInstance().getExternalContext().getUserPrincipal().getName()
```

To determine whether a user is in a specific role, the following boolean statement can be used:

```
FacesContext.getInstance().getExternalContext().isUserInRole("roleName");
```

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 (Lightweight Directory Access Protocol) 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 used, while in other environments such as test- and production environments, a full-blown security implementation such as LDAP can be implemented, again without any changes to the application code.

When using Oracle's web container OC4J, the JAAS standard is implemented using the Oracle AS JAAS Provider, shortened as "JAZN". JAZN provides out-of-the-box support for storing the user and roles information in two formats:

- In a simple XML file, typically named jazn-data.xml
- In an LDAP directory. Oracle's Internet Directory (OID) is a popular implementation of the LDAP protocol.

In addition, JAZN can be configured to authenticate users against Oracle's Single Sign-On Server (SSO).

To use JAAS-JAZN with JHeadstart, you set the **Authentication Type** property to "JAAS".

If you want to use role-based authorization using JAAS, then check the **Use Role-Based Authorization?** Checkbox and set the **Authorization Type** to "JAAS". This only makes sense if your LDAP directory contains a useful role structure that is linked to your application users stored in LDAP. If no role information is present in LDAP, or it is too coarse-grained to be of use for the authorization levels that need to be applied in your application, then set the **Authorization Type** to "Custom", so you can use the JHeadstart security tables or your own tables to implement authorization. If LDAP contains useful role information, and you want to use additional application-specific roles, then set the **Authorization Type** to "JAAS and Custom".

Warning: All application roles need to be listed in the web.xml deployment descriptor for the isUserInRole() API call to work properly. In other words, your list of application roles needs to be maintained in two places, in the LDAP accessed by JAAS, and in the web.xml.

If you need to deploy your ADF-JHeadstart application to another web container, like JBoss, Tomcat, or Websphere, you need to consult the documentation of this web container for information on configuring JAAS and availability of out-of-the-box JAAS providers like OracleAS JAAS (JAZN).

Overview of JAAS-based security in OC4J. OC4J Security Guide, chapter 2
http://download-uk.oracle.com/docs/cd/B25221_03/web.1013/b14429/jaas_intro.htm

10.1.2. JAAS Custom Login Module

JAAS supports the concept of a **Custom Login Module (CLM)**. A CLM allows you to retrieve security information from an arbitrary information store, for example a set of database tables. If none of the standard JAAS providers of your web container meets your requirements, you can write a custom login module. Since accessing database tables for obtaining the security information is a common use case for a custom login module,

OC4J ships with a ready-to-use sample CLM that you can configure to access your own security tables.

Using Custom Login Modules in OC4J.
http://iasdocs.us.oracle.com/iasdl/101310_final/web.1013/b28957/loginmod.htm

Other examples of Database Login Modules. If you cannot use OC4J to deploy your web container, then the sample database login modules written by Frank Nimphius and Duncan Mills might be helpful since they do not rely on OC4J-specific classes. This OTN article describes how to use these login modules:
http://www.oracle.com/technology/products/jdev/howtos/10g/jaassec/index.htm

Although a custom login module provides a lot of flexibility in storing the security information, your flexibility is limited because of the following characteristic of the use of JAAS in web applications:

Warning: All application roles need to be listed in the web.xml deployment descriptor for the isUserInRole() API call to work properly. In other words, your list of application roles needs to be maintained in two places, in your custom security information store accessed through the CLM, and in the web.xml.

To use a JAAS CLM with JHeadstart, you set the **Authentication Type** property to "JAAS Custom Login Module". If the CLM also retrieves user role information, then check the Use Role-based Authorization checkbox, and set **Authorization Type** to "JAAS".

10.1.3. Hardcoding Roles or Permissions in Application Code

When you add security to your application, you need to check whether a user is authorized to perform some application function. A fundamental choice you have to make when implementing application security is how to perform these checks. The most obvious choice is by checking the user's roles. This implies that you need to design the role structure of your application upfront, and your application code will contain hard-coded role names to perform the various security checks.

Once in production, administration of application security is limited to assigning the proper roles to your application users. Although you might have defined the application roles in database tables for easy administration, adding new roles requires changes to the application code, adding hardcoded references to the new application role. In other words, changing the security schema of your application always requires a new release of your application code.

A less obvious but more flexible approach is by using the concept of *permissions*. Each application function you want to secure can be defined as a permission. A permission is granted to one or more application roles, which in turn can be granted to one or more users. For example, in the context of JHeadstart, you might think of the following group permissions:

- The "Jobs" permission provides access to the pages generated from the Jobs group as defined in the Application Definition Editor. Without this permission, the user will not see the Jobs menu entry, and will get an access denied error if he tries to access the page by "hacking" the URL in the browser.
- The "Jobs.Create" permission determines whether the "New Job" button is rendered.

- The “Jobs.Update” permission determines how the items on the Edit Job page will be rendered: as read only when the user does not have a role with this permission, or as updateable when the user does have this permission.
- The “Jobs.Delete” permission determines whether the “Delete Job” button is rendered.

By hardcoding permission names in application code, the security schema of your application is fully configurable at runtime, new roles can be added and existing roles can be deleted or changed by adding or removing permissions, without changes to the application code. Adding new permissions still requires a new application code release, but this release was needed anyway to add the new application function that is to be secured by the new permission.

To use permission-based security with JHeadstart, you check the service-level checkbox **Authorize Using Group Permissions?**.

JHeadstart comes with a set of database tables to store the role and permission information, and an administration application that can be used to maintain this information. When used in combination with JAAS-based authorization, JHeadstart performs the authorization check by first looking up all the roles that provide access to a permission and then make the `isUserInRole()` API call to check whether the user has access to such a role.

10.1.4. Custom Security

Instead of using JAAS, you can use the JHeadstart custom security mechanism. This is implemented in application logic using a servlet filter that redirects to a login page when an unauthenticated user tries to access a secured application. When the user submits the login page, custom logic fires to authenticate the user and to retrieve the user’s roles and, optionally, permissions.

Historically, 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 Business Service layer of the application.

Note however that by using a JAAS Custom Login Module, the same database tables could be accessed. The only drawback of using JAAS is the double maintenance of role information; new roles must be added to the web.xml as well.

10.1.5. ADF Model Security

ADF allows you to secure the executables (iterator bindings), action and value bindings defined in the ADF page definitions, part of the ADF Model layer (also known as ADF Data Bindings layer).

This provides additional security on top of the security defined in the View layer. If, for example the user is now allowed to create a new employee, the “New Employee” button is typically hidden in the user interface. Using ADF Model Security, an additional check can be added on the “CreateEmployee” action binding in the page definition. Should the user somehow manage to submit an HTTP request to create a new employee, the ADF Model layer will prevent this and raise an application error as the user does not have the privilege to execute the “CreateEmployee” action binding.

Similarly, read and update privileges can be defined on value bindings. If a user tries to update a UI item that is based on a value binding on which he does not have update privilege, the ADF Model layer will raise an application error. Note that with ADF Faces as view technology, this error is never raised, since the input control will be already rendered as read only because ADF Faces checks whether the underlying value binding is updateable.

Introduction into ADF Security. OTN article that provides a good overview
http://www.oracle.com/technology/products/jdev/howtos/1013/adfsecurity/adfsecurity_10132.html

ADF Developer's Guide, Implementing Authorization Using Oracle ADF Security.
http://download-west.oracle.com/docs/html/B25947_01/adding_security007.htm

Note that if you enable ADF Security, you will need to grant privileges to each and every binding in each and every page definition. By default, once ADF security is enabled, the user does not have access to any binding. JHeadstart does NOT generate security settings on the page definition bindings!

ADF Model Security relies on JAAS. The ADF Security wizard will configure J2EE security in the web.xml, similar to JHeadstart, but less powerful. It does not use a JSF login page, and does not create sample users and roles. When you run the ADF Security wizard after you generated JAAS-based authentication with JHeadstart, step 4 of the wizard will pre-display the form-based authentication settings generated by JHeadstart as shown in the screen shot below.

Unfortunately, due to a bug in the wizard validation code that does not accept the Login Page and Error Page URL's, you will get an error message when you try to proceed to the next page. The work around is as follows:

- Check the **Generate Default** checkbox and finish the ADF Security wizard.
- Go to the web.xml, and remove the < login-config> element, and "AllPages" <security-constraint> element.

- Regenerate your application again using JHeadstart. This will move back the JHeadstart-generated <login-config> element to the web.xml, which was overridden when finishing the ADF Security wizard.

10.1.6. ADF BC Security

When using ADF Business Components, you can define Read-Only, Update, and Update-While-New permissions on an entity object, or on individual attributes of an entity object.

Similar permissions can be set on the page definitions using ADF Model Security, if used both, the most restrictive permission wins. Using ADF BC Security is more efficient in configuring. Authorization rules are defined once at the entity object level and are automatically carried forward to all view objects based on these entity objects, and to all page definitions that contain iterator and value bindings based on these view objects.

ADF BC Security relies on JAAS. If you want to use it, you need to configure JAAS-based security with JHeadstart.

ADF Developer's Guide, Configuring the ADF Business Components Application to Use Container-Manage Security.
http://download-uk.oracle.com/docs/html/B25947_01/adding_security004.htm

10.1.7. ADF Model Security vs. ADF BC Security

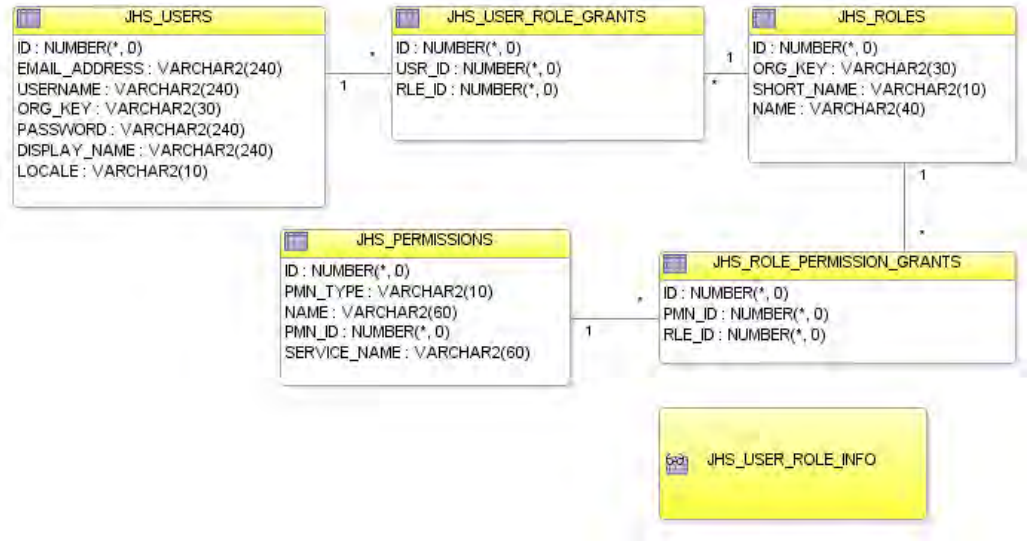
The added value of ADF Model Security on top of ADF BC security is the possibility to declaratively secure action/method bindings. This comes at the price of much more overhead in configuring the authorization rules. You cannot use ADF Model Security to *only* secure action/method bindings, once enabled you need to specify authorization rules for all bindings.

So, as an alternative, you might choose to programmatically add a security check in Java at the start of the methods underlying the action/method bindings.

If your application accesses multiple business service technologies, not just ADF Business Components, ADF Model Security becomes a more attractive option, in particular when those other technologies do not have the built-in authorization support provided by ADF BC.

10.2. JHeadstart Security Tables and Security Administration Screens

JHeadstart uses a set of database tables to support some of the security options discussed in the previous paragraph. The structure of these tables is shown below.

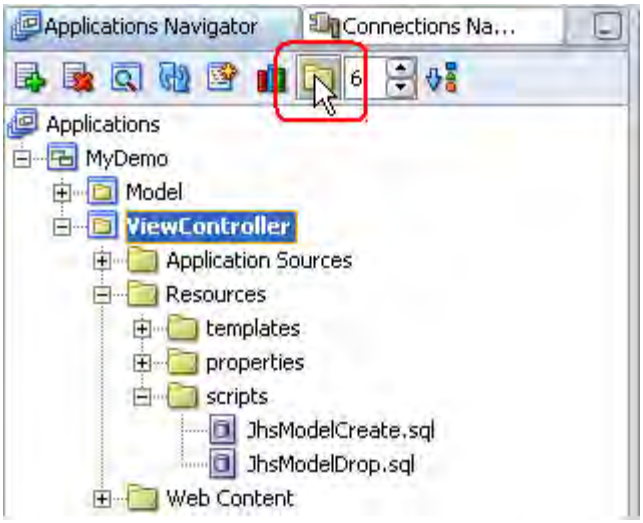


Depending on the security settings you have made, you will use all, some or none of the above database objects:

- When **Authentication Type** is set to "JAAS", and **Authorization Type** is set to "JAAS" and checkbox **Authorize Using Permissions** is unchecked, then none of the above database objects is used.
- When **Authentication Type** is set to "Custom", the JHS_USERS table is used to authenticate the user.
- When **Authentication Type** is set to "JAAS with Custom Login Module", the JHS_USERS, JHS_USER_ROLE_GRANTS and JHS_ROLES tables are used, as well as the JHS_USER_ROLE_INFO database view.
- When **Authorization Type** is set to "Custom" or "JAAS and Custom", the JHS_USERS, JHS_USER_ROLE_GRANTS and JHS_ROLES tables are used
- When checkbox **Authorize Using Group Permissions** is checked, the JHS_ROLES, JHS_ROLE_PERMISSION_GRANTS and JHS_PERMISSIONS tables are used.

10.2.1. Creating the Database Tables

You create the above database objects by running the script JhsModelCreate.sql against the database connection of your application schema. This script is located in the scripts directory of your ViewController project. If you don't see the scripts directory, make sure you click the Toggle Directories in the toolbar of the Application Navigator.



You can right-mouse-click on the JhsModelCreate.sql, then choose Run in SQL*Plus, and then the database connection you want to run the script in.

☞ **Attention:** We recommend installing the JHeadstart tables in the same schema as your own application tables. If you nevertheless prefer to install the JHeadstart tables in a different database schema, then you need to ensure that your application schema has full access to the JHeadstart tables and synonyms with the same name as the table name. This is required because the JHeadstart runtime accesses the database tables through View Object usages defined in application module **JhsModelService**. When generating your application while using one or more of the table-driven features, this JhsModelService application module is added as a nested usage to your own application module, thereby "inheriting" the database connection of its parent application module.

☞ **Attention:** The JhsModelCreate.sql script creates database tables for all table-driven JHeadstart runtime features. Additional tables for dynamic menus, translations and flex items are also created. If you do not plan to use these other features you can create your own script that only creates the above tables and view, and the JHS_SEQ sequence that is used to populate the ID column in these tables.

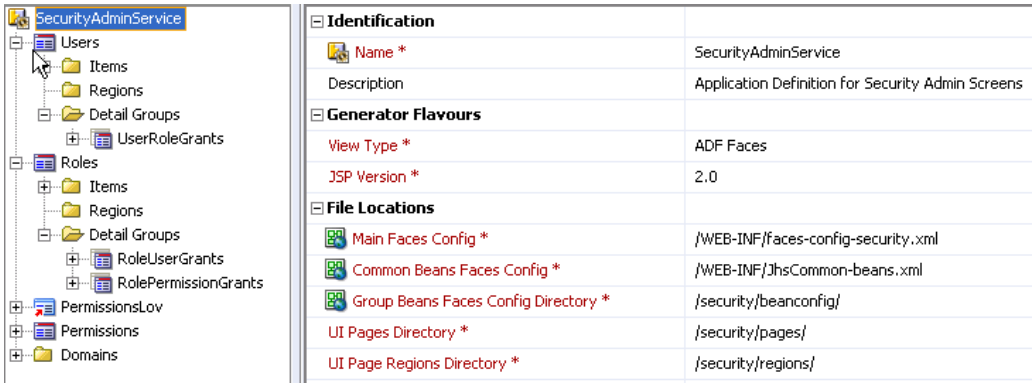
You can use your own security tables rather than the JHeadstart tables, if you prefer so. See section Using Your Own Security Tables for more information.

The JHeadstart runtime includes predefined "hooks" where you can plug in your own security code to access your own security tables. The hooks to use depend on your security settings, and will be described in the next paragraphs.

10.2.2. Generating Security Administration Pages

If you generate your application with security settings that use one or more JHeadstart database tables (see above), then as part of the generation run, a separate application definition SecurityAdminAppDef.xml is generated that can be used to generate the security administration screens. This application definition is only generated when it does not exist yet, so after it has been generated, you can make any changes you want using the Application Definition Editor, without loosing these changes when you regenerate your "own" application definition.

Before you generate the SecurityAdminAppDef you might want to inspect the file locations properties, and change them if you have set other naming standards for these properties.



The default settings are in line with what we recommend:

- Generate all Main faces-config files in the WEB-INF directory with a suffix indicating the service
- JhsCommon-beans.xml is shared by all services (application definitions); the location should never be changed.
- Create a subdirectory per service for the generation of group beans faces config files, pages and regions. Make a subdirectory under this "service" directory for each of these file types.
- The **View Package** property should be the same across all services
- The **Resource Bundle Type** and **NLS Resource Bundle** properties should be the same across all services.

Note that all service-level properties that are generally the same across all services have got the same values as in your "own" application definition. If you later on decide to make changes to these settings in your own application definition, then you will need to make the same change in the SecurityAdminAppDef Application Definition.

As you can see in the above screen shot, the generated SecurityAdminAppDef file contains groups to maintain all JHeadstart security tables. Depending on your security settings, you may delete groups that maintain tables you will not use:

- You can remove the Users, UserRoleGrants and RoleUserGrants groups if you use **Authentication Type** "JAAS" and **Authorization Type** "JAAS".
- You can remove the PermissionsLov, Permissions, and RolePermissionGrants groups if you unchecked the **Authorize Using Group Permissions** checkbox.

Now, if you are satisfied with the settings, you can run the JAG for the SecurityAdminAppDef.

10.3. Using JAAS-JAZN for Authentication

When you run the JHeadstart Application Generator with service-level property **Authentication Type** set to "JAAS", the following happens:

- A login page and associated login bean is generated.
- A logout button and logout bean is generated.
- J2EE security is set up in the web.xml.
- Default users and roles are defined in jazn-data.xml.

These actions are discussed below in more detail.

10.3.1. Login Page and Login Bean

An ADF Faces login page is generated in `/security/pages` subdirectory under the html root directory. This file is generated through the `default/misc/file/fileGenerator.vm` template, which in turn uses `default/misc/file/loginPage.vm` template. The login page is only generated when it does not exist yet, so you can customize the generated login page without loosing these changes when regenerating.

When clicking the login button on the login page, the `authenticateUser` method of the generic `oracle.jheadstart.controller.jsf.bean.LoginBean` class is called. This bean is configured in JhsCommonBeans.xml. In case of JAAS authentication, this method redirects to a J2EE login form which autosubmits itself, and is therefore not visible to the user. The J2EE login form contains the required form action `j_security_check`, and fields `j_username` and `j_password`, filled with the values as entered in the ADF Faces login page, to trigger the J2EE container-managed security.

Using this "redirect" technique, we are able to use a normal JSF page as login page, so you can apply the same ADF Faces look and feel as used by your other application pages, and you can use ADF drag and drop data binding should you want to add dynamic data to the login page, like news items read from a database table.

The generated login page contains two "fast login" links for users SKING and AHUNOLD, the two sample users that are created in the jazn-data.xml file.

10.3.2. Logout Button and Logout Bean

Using the /default/misc/file/menuGlobal.vm template, called from the default/misc/file/fileGenerator.vm template, a logout button is generated in the global buttons area. When clicking the logout button, the `logout` method of the generic `oracle.jheadstart.controller.jsf.bean.LogoutBean` class is called. This bean is configured in JhsCommonBeans.xml. In this method, the session is invalidated and a redirect to the logout destination URL is performed, which defaults to "/". By using the slash, the web container will launch the index.jsp page that JHeadstart generated in the HTML root directory. The index.jsp page redirects to the generated home page, causing the login page to appear first again, but you are free to change the redirect destination in the index.jsp page.

10.3.3. J2EE Security Set Up in web.xml

JHeadstart first checks whether the <login-config> element is present in the web.xml. If it is not present, then the following is generated:

- A <login-config> element is generated which configures the web container to use the generated login page.
- A <security-constraint> element is generated which provides access to all .jspx pages for users that have the Administrator role or the User role. The actual names of these roles are defined in the service level properties **Administrator Role** and **User Role**.
- For both roles, a <security-role> element is also generated.

The sample roles and security constraint are set up as a convenience, just like the links on the login page, they allow you to access the generated application as administrator and normal user.

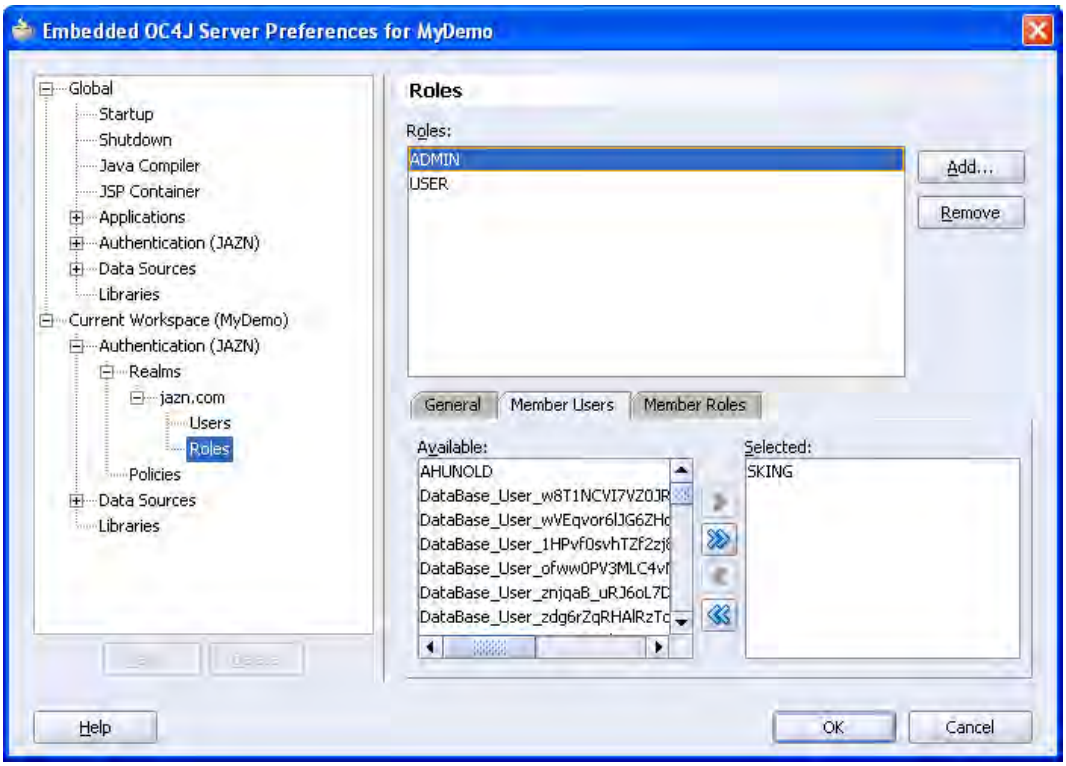
You can customize these security elements to your liking. As long as the <login-config> element is present in the web.xml, JHeadstart will preserve your customized settings when regenerating your application.

10.3.4. Default Users and Roles in jazn-data.xml

JHeadstart generates the jazn-data.xml file in the META-INF directory under the source root directory. It also generates the <jazn> element in orion-application.xml, if not present yet, to configure OC4J to use the jazn-data.xml file as security information provider:

```
<jazn provider="XML" location="./jazn-data.xml"/>
```

The jazn-data.xml specifies two default users, SKING and AHUNOLD with passwords the same as the username, and two roles, the Administrator role assigned to SKING and the user role assigned to AHUNOLD. You can customize the default users and roles either directly in the jazn-data.xml file, or using the editor provided by JDeveloper, which can be found under the Tools -> Embedded OC4J Preferences menu.



Note that if you use the JDeveloper editor, the modifications to jazn-data.xml are not saved until you close JDeveloper.

You can customize the default users and roles to your liking. JHeadstart will preserve your changes upon regeneration.

10.3.5. Using LDAP and/or Single Sign On in Deployed Application

The users and roles generated in jazn-data.xml facilitate testing in the development environment using JDeveloper's embedded OC4J or a stand-alone OC4J server. When you deploy your application to a test or production environment, you will typically change the security provider to an LDAP implementation, either Oracle's Internet Directory (OID), or an external LDAP Provider. You can do this after deploying your application by using the OracleAS or OC4J stand-alone Management console, or you can change the <jazn> settings in orion-application.xml before deploying your application. See the OC4J Security Guide for more information.

OC4J Security Guide. Chapter 2 Overview of OC4J Security, Chapter 6 Oracle Identity Management Security provider.
http://download-uk.oracle.com/docs/cd/B25221_03/web.1013/b14429/jaas_intro.htm

10.4. Using JAAS with Custom Login Module for Authentication

When you run the JHeadstart Application Generator with service-level property **Authentication Type** set to "JAAS with Custom Login Module", the following happens:

- A login page and associated login bean is generated. See section Login Page and Login Bean for more information.
- A logout button and logout bean is generated. See section Logout Button and Logout Bean for more information.
- J2EE security is set up in the web.xml. See section J2EE Security Set Up in web.xml for more information.
- The SecurityAminAppDef application definition file is generated. See section Generating Security Administration Pages for more information.
- SQL script createSampleUsersAndRoles.sql is generated.

10.4.1. Sample Users And Roles

Using the default/misc/file.createSampleUsersAndRoles.vm template, launched from the default/misc/file/fileGenerator.vm template, the SQL script createSampleUsersAndRoles.sql is generated into the /scripts directory when it does not exist yet. It is automatically executed as well when service-level property **Run Generated SQL Scripts?** is checked. The script creates two users in JHS_USERS table, SKING and AHUNOLD, two roles in JHS_ROLES table, the administrator role as specified in the **Administrator Role** property, and the user role as specified in the **User Role** property. SKING is assigned the administrator role, AHUNOLD the user role through two entries in the JHS_USER_ROLE_GRANTS table.

10.4.2. Configuring the Custom Login Module

Unlike other security settings, using a JAAS Custom Login Module requires additional manual steps from you, the developer, before you can run your application using the embedded OC4J. The embedded OC4J is configured differently than the standalone OC4J to support runtime testing of applications without requiring application deployment. To do this, all web applications are executed as "current-workspace-app", no matter what the assigned name for the J2EE application. This information is important because to use JAAS Login Modules with the embedded OC4J, they need to be configured under the name of the application using it: **current-workspace-app.** Failing to use the **current-workspace-app** name for the LoginModule will cause OC4J to use its own default Realm LoginModule and look for the username / password pair in the system-jazn-data.xml file directly. Thus, failing to find login credentials would end in an unauthenticated request.

All configuration files of the embedded OC4J are located in the <jdev_home\jdev\system\oracle.j2ee.10.1.3.xx.xx\embedded-oc4j\config directory where xx.xx is replaced with the actual build number. The configuration files that you will edit to configure the Custom Login Module are:

- system-jazn-data.xml
- application.xml
- data-sources.xml

Note that the changes you make in these files are applied globally, they apply to all applications that you run in the embedded OC4J. If you are simultaneously developing another application that does not use JAAS, you need to comment out again the changes described below.

10.4.3. System-jazn-data.xml

The system-jazn-data.xml file contains the JAAS LoginModule definition. We recommend using the `DBTableOraDataSourceLoginModule` which is shipped with OC4J, already exists in the class path of OC4J and doesn't need to be added to the lib directory or configured in the application.xml file. It is ready for use. In system-jazn-data.xml, you can specify so-called loginmodule options to configure the `DBTableOraDataSourceLoginModule` to use the JHeadstart security tables. The login module does have some limitations though:

- The username column must be the primary key column in the users table. This is not the case in the JHS_USERS table, which has a meaningless ID column as primary key
- The users table can be joined with one other table to retrieve the role names of the user. In the JHeadstart security model, we need to join with two other tables, JHS_ROLES to get the role name, and JHS_USER_ROLE_GRANTS to link the user to the granted roles.

To work around these limitations, we use the JHS_USER_ROLE_INFO view, which is defines as follows:

```
create or replace view jhs_user_role_info
as
select rle.id
      ,rle.short_name
      ,rle.name
      ,usr.username
from   jhs_roles rle
      ,jhs_user_role_grants urg
      ,jhs_users usr
where  urg.rle_id = rle.id
and    urg.usr_id = usr.id
```

With this view in place, we need to add the following to system-jazn-data.xml (don't forget to change the data_source_name, see below):

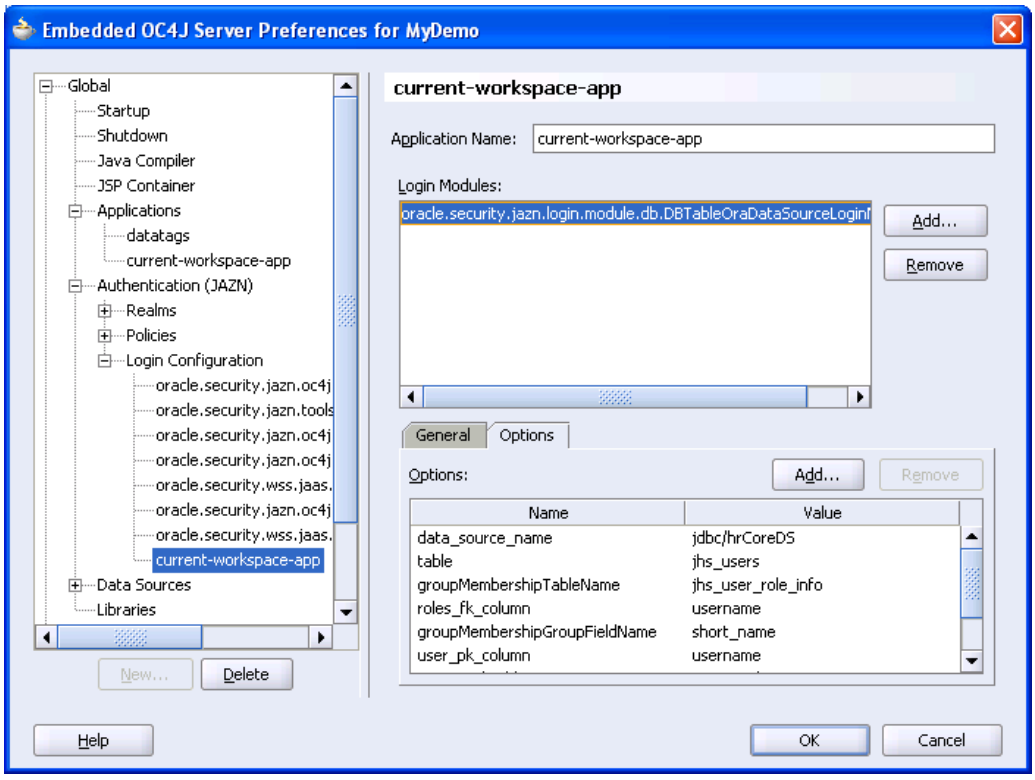
```
 <application>
  <name>current-workspace-app</name>
   <login-modules>
       <login-module>

<class>oracle.security.jazn.login.module.db.DBTableOraDataSourceLoginModule</class
>
         <control-flag>required</control-flag>
         <options>
           <option>
             <name>data_source_name</name>
             <value>jdbc/hrCoreDS</value>
           </option>
           <option>
             <name>table</name>
             <value>jhs_users</value>
           </option>
           <option>
             <name>groupMembershipTableName</name>
             <value>jhs_user_role_info</value>
           </option>
           <option>
             <name>roles_fk_column</name>
             <value>username</value>
           </option>
           <option>
```

```
            <name>groupMembershipGroupFieldName</name>
            <value>short_name</value>
          </option>
          <option>
            <name>user_pk_column</name>
            <value>username</value>
          </option>
          <option>
            <name>passwordField</name>
            <value>password</value>
          </option>
          <option>
            <name>usernameField</name>
            <value>username</value>
          </option>
          <option>
            <name>casing</name>
            <value>sensitive</value>
          </option>
        </options>
      </login-module>
    </login-modules>
  </application>
```

The value of the data_source_name property should match the datasource auto-created by JDeveloper for your database connection. The naming format of the data source is

```
jdbc/[DBConnectionName]CoreDS
```

While nothing prevents you from hand editing the system-jazn-data.xml file, JDeveloper provides a dialog to assist you. To launch the editor, select the menu option Tool -> Embedded OC4J Server from the JDeveloper toolbar.



10.4.4. Application.xml

In the application.xml, the <jazn> element should be changed as follows:

```
<jazn provider="XML">
   <property name="role.mapping.dynamic" value="true"/>
   <property name="custom.loginmodule.provider" value="true"/>
```

```
</jazn>
```

The `role.mapping.dynamic` property defines that J2EE security roles should be read from the authenticated user subject, allowing the LoginModule to add role grants to the authenticated user. The `custom.loginmodule.provider` property tells OC4J to use a custom JAAS LoginModule for authentication.

10.4.5. Debugging the Custom Login Module

You can easily make a typo in the configuration, or use a non-existent data source name, in all these situations the following error is logged, which you also get when entering an invalid username or password:

```
WARNING: Login Failure: all modules ignored
javax.security.auth.login.LoginException: Login Failure: all modules ignored
    at javax.security.auth.login.LoginContext.invoke(LoginContext.java:921)
    at javax.security.auth.login.LoginContext.access$000(LoginContext.java:186)
    at javax.security.auth.login.LoginContext$4.run(LoginContext.java:683)
    at java.security.AccessController.doPrivileged(Native Method)
    at javax.security.auth.login.LoginContext.invokePriv(LoginContext.java:680)
    at javax.security.auth.login.LoginContext.login(LoginContext.java:579)
    at oracle.security.jazn.oc4j.OC4JUtil.doJAASLogin(OC4JUtil.java:241)
    ….
```

To discover what is really wrong, it is helpful to turn on debugging. You can do this by editing the j2ee-logging.xml file in the config directory of your embedded OC4J, and change the line

```
<logger name="oracle" level="NOTIFICATION:1" useParentHandlers="false">
```

to

```
<logger name="oracle" level="FINEST" useParentHandlers="false">
```

When you run the application again with this setting, and assume you specified an invalid data source name, you will get feedback like this:

```
…
Oct 7, 2007 10:53:26 PM
oracle.security.jazn.login.module.db.DBTableOraDataSourceLoginModule
performDbAuthentication
FINE: [DBTableOraDataSourceLoginModule]Error: jdbc/hrCoreCitroenDS not found
Oct 7, 2007 10:53:26 PM
oracle.security.jazn.login.module.db.DBTableOraDataSourceLoginModule login
FINE: [DBTableOraDataSourceLoginModule]Logon Successful = false
```

10.4.6. Deploying your Application with Custom Login Module

When you deploy your application to a test or production environment, you can either configure your security provider after deployment using the OracleAS or OC4J management console, or you can include the <jazn-loginconfig> element in your orion-application.xml to auto-configure the security provider when deploying your application.

If you choose the latter, you can copy and paste the content of the <jazn-loginconfig> element in system-jazn-data.xml to your orion-application.xml, but don't forget to replace the application name **current-workspace-app** with the actual name used to deploy your application.

The <jazn> properties you added in application.xml are already generated into your orion-application.xml.

10.5. Using Custom Authentication

When you run the JHeadstart Application Generator with service-level property **Authentication Type** set to "Custom", the following happens:

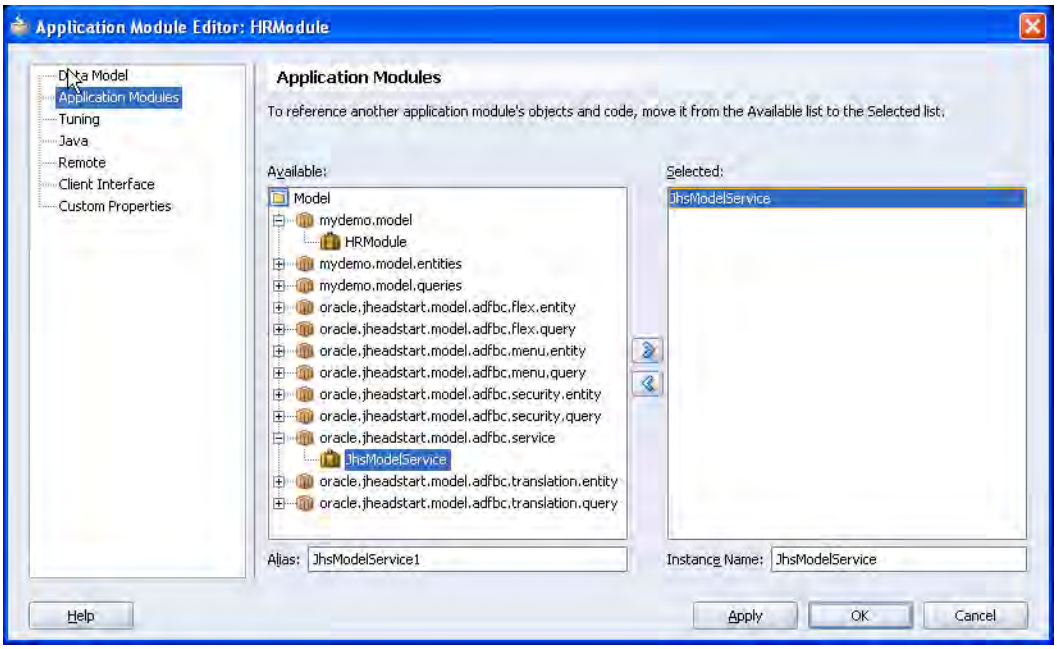
- The JHeadstart authentication servlet filter is configured in the web.xml
- JhsModelService application module is added as a nested application module to your application module.
- A login page and associated login bean is generated.
- A logout button is generated
- SQL script createSampleUsersAndRoles.sql is generated. See section Sample Users And Roles for more information.
- The SecurityAminAppDef application definition file is generated. See section Generating Security Administration Pages for more information.

10.5.1. JHeadstart Authentication Filter

The JHeadstart runtime includes servlet filter class `oracle.jheadstart.controller.jsf.AuthenticationFilter`. This servlet filter is configured in the web.xml to ensure that the user is redirected to the login page when the user is not yet logged in. The servlet filter also supports logout, by invalidating the session and redirecting to the logout destination URL which defaults to "/". By using the slash, the web container will launch the index.jsp page that JHeadstart generated in the HTML root directory. The index.jsp page redirects to the generated home page, causing the login page to appear first again, but you are free to change the redirect destination in the index.jsp page.

10.5.2. Nested JhsModelService Application Module

All ADF Business Components included in the JHeadstart Runtime library are imported into your Model Project, and the **JhsModelService** application module is added as a nested usage to your own application module. The **JhsModelService** includes View Object Usages that insert, update, delete and query the underlying database tables needed for the table-driven JHeadstart features, including authentication. Note that by creating JhsModelService as a nested application module, it will inherit the database connection of the parent application module.



10.5.3. Login Page and Login Bean

An ADF Faces login page is generated in /security/pages subdirectory under the html root directory. This file is generated through the default/misc/file/fileGenerator.vm template, which in turn uses default/misc/file/loginPage.vm template. The login page is only generated when it does not exist yet, so you can customize the generated login page without loosing these changes when regenerating.

When clicking the login button on the login page, the `authenticateUser` method of the generic `oracle.jheadstart.controller.jsf.bean.LoginBean` class is called. This bean is configured in JhsCommonBeans.xml. In case of custom authentication, this method calls method `authenticateUser(String username, String password)` on the nested JhsModelService application module. This method uses a ViewObject to access the JHS_USERS table to validate the username and password. When valid, the method returns the `UserContext` object that implements the `JhsUser` interface. This object is stored on the session using "JhsUser" as key, which is checked by the authentication filter to determine whether the user is already logged in.

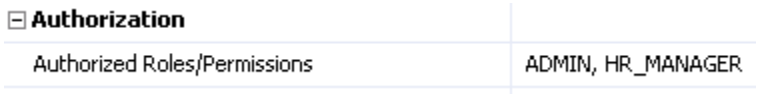
The generated login page contains two "fast login" links for users SKING and AHUNOLD, the two sample users that are created in SQL script createSampleUsersAndRoles.sql.

10.5.4. Logout Button

Using the /default/misc/file/menuGlobal.vm template, called from the default/misc/file/fileGenerator.vm template, a logout button is generated in the global buttons area. When clicking the logout button, you navigate to a non-existent page using the URI /faces/security/pages/Logout.jspx. However, this URI is configured in the JHeadstart Authentication filter as the logout URL, see section JHeadstart Authentication Filter for more information.

10.6. Restricting Access to Groups based on Authorization Information

When you have checked the service-level checkbox **Use Role-based Authorization** and you selected an **Authorization Type**, you can restrict access to the pages generated for each group by specifying roles or permissions. This can be done using the **group level** property **Authorized Roles/Permissions** where you can specify a comma-separated list of roles and/or permissions. If the user is granted at least one of the roles or permissions, he is authorized to access the page.



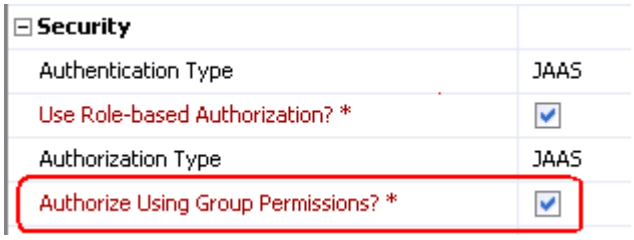
If this property is not set, the pages generated for this group are public, and do not require a specific user role or permission.

If you protect group pages using this property, JHeadstart will implement this restriction in both the View and Controller layer:

- **View layer (JSF pages)**: Hide tabs and navigation buttons that go to a page of that group if the currently logged-in user is not authorized. See section JHeadstart Authorization Proxy for more information on how this is implemented.
- **Controller Layer**: If the user tries to directly access an unauthorized page by "hacking" the browser URL, he should still be denied access. JHeadstart performs this check for you. See section JHeadstart Authorization Proxy for more information on how this is implemented.

10.6.1. Restricting Group Access using Permissions

The above example used role names to restrict access to a particular group. As explained in section Hardcoding Roles or Permissions in Application Code you might prefer to authorize using permission names. To do so, you check the service-level checkbox **Authorize Using Group Permissions**.



Note that this property can be used regardless of the values set for **Authentication Type** and **Authorization Type**.

When you generate your application with this setting, the following happens

- All groups are protected using the group name as permission name. This means that you do not have to specify the **Authorized Role/Permissions** property for each and every group. You can still specify this property at the group level, to override the default group name permission.

- A SecurityAdminAppDef application definition file is generated that can be used to generate pages to administer the permissions and grant permissions to roles. Using a multi-select List of Values, you can easily search and assign multiple permissions to a role. See section Generating Security Administration Pages for more information.
- A SQL script named PermissionsData[ServiceName].sql is generated in the /scripts directory. The script is automatically executed when service-level checkbox **Run Generate SQL scripts?** is checked. The script inserts entries in JHS_PERMISSIONS table for each group. Four permissions are inserted for each group, an access permission named after the group, and three "operation" permissions for creating, updating and deleting. See section Restricting Group Operations based on Authorization Information for more information on using these operation permissions. In the same script, all permissions are granted to the Administrator role as specified in the **Administrator Role** property. This means that you when you use the sample user SKING to log in, you should still be able to access all groups. If you log in as AHUNOLD you will get an access denied message since the USER role does not have any permissions granted. You can use the security administration application to grant permission privileges to the USER role, as shown in the screen shot below. After you granted permissions for one or more groups, and you will log in as AHUNOLD you will see the group tabs for which you granted access permission. Depending on the group action permissions granted, the group pages will allow for insert, update and/or delete.

10.6.2. When Access Denied Go To Next Group

Suppose you have combined several Application Definitions into a single application by providing links to the starting points of each Application Definition. That starting point would be the first top-level group of the Application Definition. Now suppose that the logged-in user does not have access to the first group. In that case you would want the link to navigate to the second group. And if the user doesn't have access to the first and second group, the link should go to the third group, etc.

JHeadstart can generate such a navigation scenario if you check the service-level property **When Access Denied go to Next Group**.

10.6.3. JHeadstart Authorization Proxy

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 and/or custom security).

To accomplish this, the JHeadstart runtime includes a class called `JhsAuthorizationProxy`. If you checked the Service-level property **Use Role-based Authorization?** in the Application Definition, a managed bean is generated into JhsCommon-beans.xml that automatically creates an instance of this class and puts it on the session.

```
<managed-bean>
  <managed-bean-name>jhsUserRoles</managed-bean-name>
  <managed-bean-class>
    oracle.jheadstart.controller.jsf.JhsAuthorizationProxy
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

This `JhsAuthorizationProxy` instance will be invoked each and every time the application needs authorization information. So whether JAAS is used and/or custom authorization mechanism, whether permission-based authorization is enabled, and whether the information is needed in the View or in the Controller layer, this 'authorization proxy' is the single point that all authorization questions are being routed through. The Authorization Proxy will determine whether standard JAAS and/or a custom security implementation is used, and will forward the 'authorization question' accordingly.

Reference: See the Javadoc of `JhsAuthorizationProxy`.

10.6.3.1. Accessing the Authorization Proxy in the View layer

For implementing security features in the View layer, for instance hiding 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 JhsAuthorizationProxy implements the Map interface. We can use the managed bean "jhsUserRoles" that was mentioned in the previous section. For instance, to hide a menu item if the current user does not belong to the 'ADMIN' or 'HR_MANAGER' roles, JHeadstart uses the following syntax:

```
<af:commandMenuItem … rendered="#{jhsUserRoles['ADMIN,HR_MANAGER']}" …/>
```

Note that you can use a comma-separated list of role and/or permission names. The Authentication Proxy will process them left-to-right until it finds a role or permission granted to the current user, and returns true in that case. If the user belongs to none of the roles, it will return false.

10.6.3.2. Accessing the Authentication Proxy in the Controller layer

As mentioned before, JHeadstart also performs a roles check in the JSF PageLifecycle. This is to prevent "URL-Hacking": the tab or button to go to a certain page might be hidden, but if the user knows the URL, he should still be denied access.

This is implemented by the method `checkRoles()` in `JhsPageLifecycle`, which is called from the `prepareModel()` phase.

This method knows which roles to check for which page, because JHeadstart generated a "roles" parameter into the Page Definition of the page.

```
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
  <parameters>
    <parameter id="roles" value="admin,manager"/>
  </parameters>
```

Reference: See the Javadoc of the methods `prepareModel()` and `checkRoles()` in the `JhsPageLifecycle` class.

10.7. Restricting Group And Item Operations based on Authorization Information

In addition to restriction group access, you can also restrict the operations based on authorization information. To do this for individual groups, use the **Insert/Update/Delete Allowed EL Expression** properties on group level.

Authorization	
Authorized Roles/Permissions	HR_MANAGER, HR_ASSISTANT
Insert Allowed EL Expression	#{jhsUserRoles['HR_MANAGER']}
Update Allowed EL Expression	#{jhsUserRoles['HR_MANAGER,HR_ASSISTANT']}
Delete Allowed EL Expression	#{jhsUserRoles['HR_MANAGER']}

In the above example, both the HR_MANAGER and HR_ASSISTANT roles can access the Employee group pages. The HR_MANAGER can insert, update and delete employee information; the HR_ASSISTANT can only update existing employees.

10.7.1. Restricting Group Operations using Permissions

As explained in section Restricting Group Access using Permissions JHeadstart can generate a SQL script that inserts operation permissions in the JHS_PERMISSIONS table for each group. These operation permissions are named after the group, suffixed with ".Create" ".Update" and ".Delete. For example, for the Jobs group the following permissions are created:

- The "Jobs.Create" permission determines whether the "New Job" button is rendered.
- The "Jobs.Update" permission determines how the items on the Edit Job page will be rendered: as read only when the user does not have a role with this permission, or as updateable when the user does have this permission.
- The "Jobs.Delete" permission determines whether the "Delete Job" button is rendered

Now, you can use these permissions rather than role names to restrict the create, update and delete operations. And you can configure this at service-level, which saves you the work of entering the **Insert Allowed**, **Update Allowed** and **Delete Allowed** EL expressions for each and every group. The same properties exist at service level, and you can use the $GROUP_NAME$ token which will be replaced with the actual group name when generating the pages for each group.

Security	
Authentication Type	JAAS
Use Role-based Authorization? *	☑
Authorization Type	JAAS
Authorize Using Group Permissions? *	☑
Role/Permission Prefix	
Administrator Role	ADMIN
User Role	USER
Insert Allowed EL Expression	#{jhsUserRoles['$GROUP_NAME$.create']}
Update Allowed EL Expression	#{jhsUserRoles['$GROUP_NAME$.upate']}
Delete Allowed EL Expression	#{jhsUserRoles['$GROUP_NAME$.delete']}
When Access Denied go to Next Group? *	☑

10.7.2. Restricting Item Operations

Based on the roles/permissions of the currently logged-in user, you can also determine if individual group items will be visible, enabled, and/or updateable.

- Visibility is determined using the properties **Display in Form Layout?**, **Display in Table Layout?**, and **Display in Table Overflow Area?**

Display Settings	
Display in Form Layout? *	#{jhsUserRoles['admin, hrmanager']}
Display in Table Layout? *	false
Display in Table Overflow Area? *	#{jhsUserRoles['admin, hrmanager']}

- Enabledness is determined using the **Disabled** property
- Updateability is determined using the **Update Allowed?** property

Operations	
Update Allowed?	#{jhsUserRoles['hrmanager']}
Disabled	#{jhsUserRoles['admin']}

Like the group access and group operations, you can also use permission names instead of role names if you enabled the option to authorize using group permissions.

10.8. Using Your Own Security Tables

You can use your own security tables rather than the JHeadstart tables, if you prefer so. The JHeadstart runtime includes predefined "hooks" where you can plug in your own security code to access your own security tables. The hooks to use depend on your security settings, as described in the next sections

10.8.1. Changes when Using JAAS Custom Login Module

Using your own security tables instead of the JHeadstart tables and view is easiest when using a JAAS custom login module: just change the options of the <jazn-loginconfig> element to reference your own table and column names. Consider creating a database view like JHS_USER_ROLE_INFO if your security data model does not comply with the prerequisites of your custom login module provider.

10.8.2. Changes when Using Custom Authentication

When you use custom authentication, the authenticateUser method of the nested JhsModelService application module is called. To hook in your own authentication logic, you should perform the following steps:

- Create your own application module that *extends* JhsModelService application module.
- Add the view object needed to authenticate the user against your own table.
- In this application module, override method authenticateUser and perform authentication using the view object created in the previous step
- Remove the JhsModelService as nested usage from your root application module
- Add your extended version of JhsModelService application module as nested usage to your root application module, **and make sure the instance name is set to JhsModelService**.

10.8.3. Changes when Using Custom Authorization and/or Permissions

The JHeadstart Authorization Proxy makes use of method

```
createUserContext(String username, String userDisplayName, boolean
addPermissionForJAASRoles)
```

on the nested JhsModelService application module. This method creates a user context object that implements the JhsUser interface, and adds authorized custom roles and permissions by calling method `setRolesAndPermissions` on the same JhsModelService application module.

So, to use your own tables for role and permission information, it is sufficient to override method setRolesAndPermissions. Override this method in your own application module that extends JhsModelService, and replace the nested JhsModelService instance with your subclass.

10.8.4. Changes to SQL Script Templates

JHeadstart uses the following templates to generate entries into the various security tables:

- default/misc/file/createSampleUsersAndRoles.vm
- default/misc/file/jhsPermissionsdata.vm

Both templates are called from the fileGenerator.vm template. So, you need to make custom copies of all three of these templates, in your custom fileGenerator.vm you can then reference your custom SQL script templates.

CHAPTER

11

Internationalization and Messaging

This chapter discusses the JHeadstart support for multiple languages, as well as the options to externalize page text strings into a (translatable) resource bundle or database table.

.

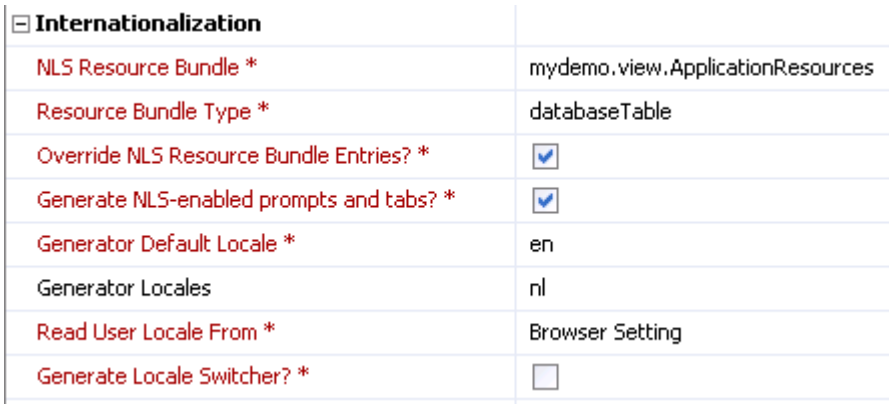
11.1. National Language Support in JHeadstart

JSF has built-in support for using property files or resource bundle classes 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.

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, a java class or a database table.

A property file is easiest to maintain by developers, it is a simple text file with key-value pairs. However, a property file does not handle special symbols well. A Java-based resource bundle is better suited for this. If you want the page text to be maintained or translated by a super user or system administrator, then using a database table as resource bundle is the best choice. See section Using Resource Bundle Type Database Table for more information.

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 Application Definition Editor will also be generated into the resource bundle.



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 optionally 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`.

11.1.1. Which Locale is Used at Runtime

The locale used to show the pages, and displayed as selected in the language drop down list is based on the property **Read User Locale From**, which defaults to the locale set in the browser of the end user.

If you do not want to read the locale from a browser, but store it as a user preference, then you can enter an JSF EL expression in the **Read User Locale From** property. You will typically use an expression that reads the locale from the user context object:

When you check the checkbox **Generate Locale Switcher**, a drop down list will be generated in the global button area which allows the end user to choose one of the supported locales (displayed through the associated language name).

11.1.2. 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
ro	Romanian
sr	Serbian
sl	Slovenian
es	Spanish

Built-in support means that if you specify one or more of these locales in the Application Definition, 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 when specifying them in the Application Definition 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.

11.1.3. Adding a non-supported Locale

If you want to generate your application using a locale that is not supported out of the box, you can do so by performing the following steps:

1. Create a new version of GeneratorText.properties for your own locale. You can find these files in the folder <ViewController project>\templates\nls.
2. Specify the locale in either the **Generator Default Locale** property or in the **Generator Locales** property.
3. Generate the application.
4. Translate the entries in the generated Resource Bundle for your locale.
5. Uncheck checkbox **Override NLS Resource Bundles** in the Application Definition, to preserve your translations. JHeadstart will then only add new keys, not change existing ones.
6. Modify <HTML Root Directory>\jheadstart\messages.js and add messages in your language. This file contains JavaScript messages.
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).

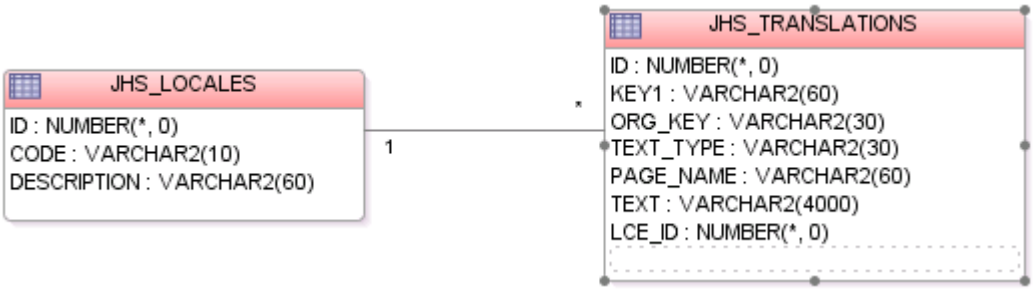
Attention: The recommended type for Resource Bundle is java instead of propertiesfile if you have special characters in your language. Make sure you compile (rebuild) the Java Resource Bundle after you have added new entries, otherwise JHeadstart Application Generator will erase them the next time you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

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/1.5.0/docs/tooldocs/solaris/native2ascii.html) can help you get the right Unicode for a specific text.

11.2. Using Resource Bundle Type databaseTable

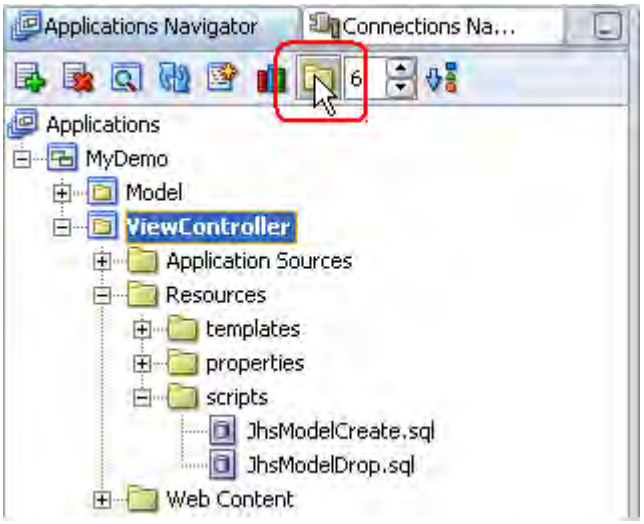
JHeadstart uses two database tables when property **Resource Bundle Type** is set to "databaseTable". The structure of these tables is shown below.



The JHS_LOCALES table contains entries for all supported locales, the JHS_TRANSLATIONS table contains all translatable strings.

11.2.1. Creating the Database Tables

You need to create the above table structure in your own application database schema. You can do this by running the script JhsModelCreate.sql against the database connection of your application schema. This script is located in the scripts directory of your ViewController project. If you don't see the scripts directory, make sure you click the Toggle Directories in the toolbar of the Application Navigator.



You can right-mouse-click on the JhsModelCreate.sql, then choose Run in SQL*Plus, and then the database connection you want to run the script in.

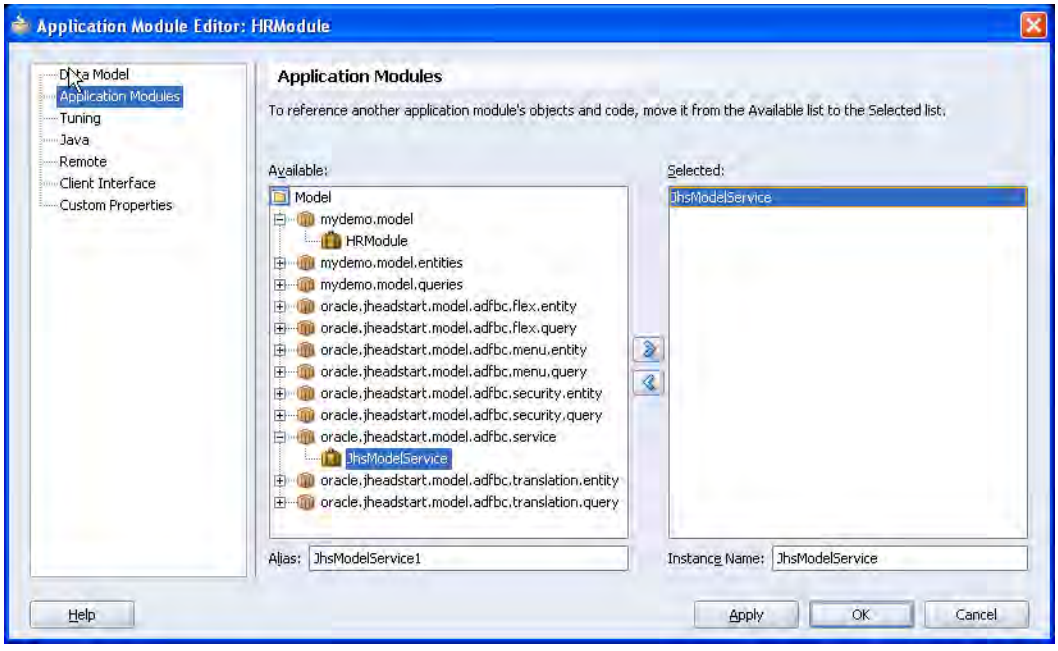
Attention: We recommend installing the JHeadstart tables in the same schema as your own application tables. If you nevertheless prefer to install the JHeadstart tables in a different database schema, then you need to ensure that your application schema has full access to the JHeadstart tables and synonyms with the same name as the table name. This is required because the JHeadstart runtime accesses the database tables through View Object usages defined in application module **JhsModelService**. When generating your application while using one or more of the table-driven features, this JhsModelService application module is added as a nested usage to your own application module, thereby "inheriting" the database connection of its parent application module.

Attention: The JhsModelCreate.sql script creates database tables for all table-driven JHeadstart runtime features. Additional tables for flex items, dynamic menus and security are also created. If you do not plan to use these other features you can create your own script that only creates the above tables, and the JHS_SEQ sequence that is used to populate the ID column in these tables.

11.2.2. Running the JHeadstart Application Generator

When you now run the JHeadstart Application Generator with property **Resource Bundle Type** set to "databaseTable", the following happens:

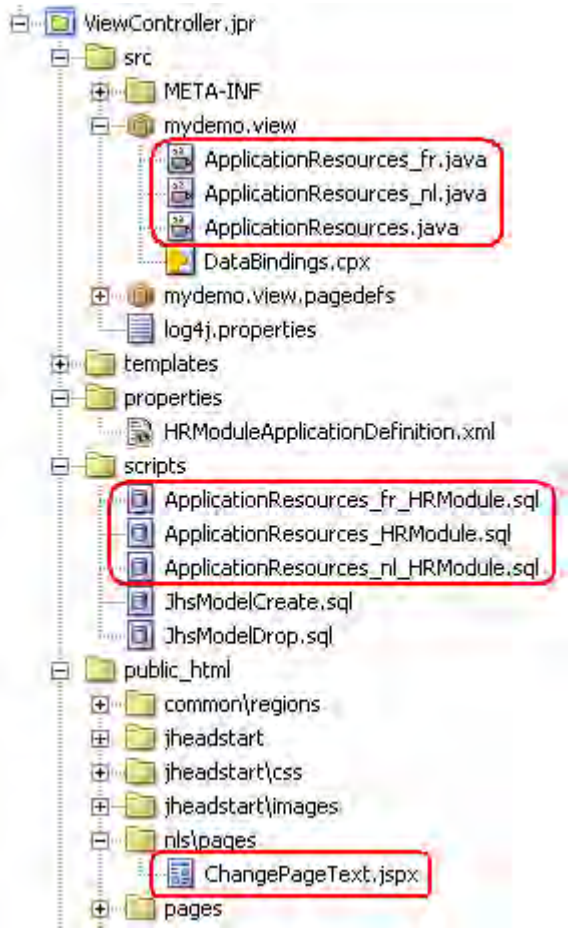
- All ADF Business Components included in the JHeadstart Runtime library are imported into your Model Project, and the **JhsModelService** application module, is added as a nested usage to your own application module. The **JhsModelService** includes View Object Usages that insert, update, delete and query the underlying nls database tables. Note that by creating JhsModelService as a nested application module, it will inherit the database connection of the parent application module.



- A SQL Script named ApplicationResources.sql is generated to insert the translatable text strings in the JHS_TRANSLATIONS table for the default locale. If not yet present, this script will also create an entry in the JHS_LOCALES table for the default locale. For each additional locale specified in the **Locales** property, a separate SQL script named ApplicationResources_*localecode*.sql is generated. The generated SQL scripts are executed automatically against the database connection of your ADF Business Components project when the service level checkbox **Run Generated SQL Scripts?** is checked.
- For the default locale, and each additional locale, a java resource bundle is generated as well. This might come as a surprise since the **Resource Bundle Type** is set to "databaseTable", not "javaClass". However, since the Java language has built-in support for resource bundle property files or java classes, and JSF integrates seamlessly with these files, we use this java class resource bundle as a "façade" to our database table. If you look at the content of the generated resource bundles, things will become more clear:

```
public class ApplicationResources
      extends TranslationTableResourceBundle
{
  public String getLocaleCode()
  {
    return "en";
  }
}
```

  The resource bundle class no longer holds the translatable strings as is the case when generating with **Resource Bundle Type** set to "javaClass". Instead it delegates retrieval of the translatable strings to the JHeadstart superclass, which uses a ViewObject in the nested JhsModelService to read the translatable text strings from the JHS_TRANSLATIONS table for the given locale.
- A page named ChangePageText.jspx is generated into the nls/pages directory under the HTML root directory. This page is a dialog page that can be used to change and translate page text in context while running the application.
- Using the default/misc/file/menuGlobal.vm template, two additional buttons are generated into the global buttons area, one button to record the page text, and one button to change/translate the page text.

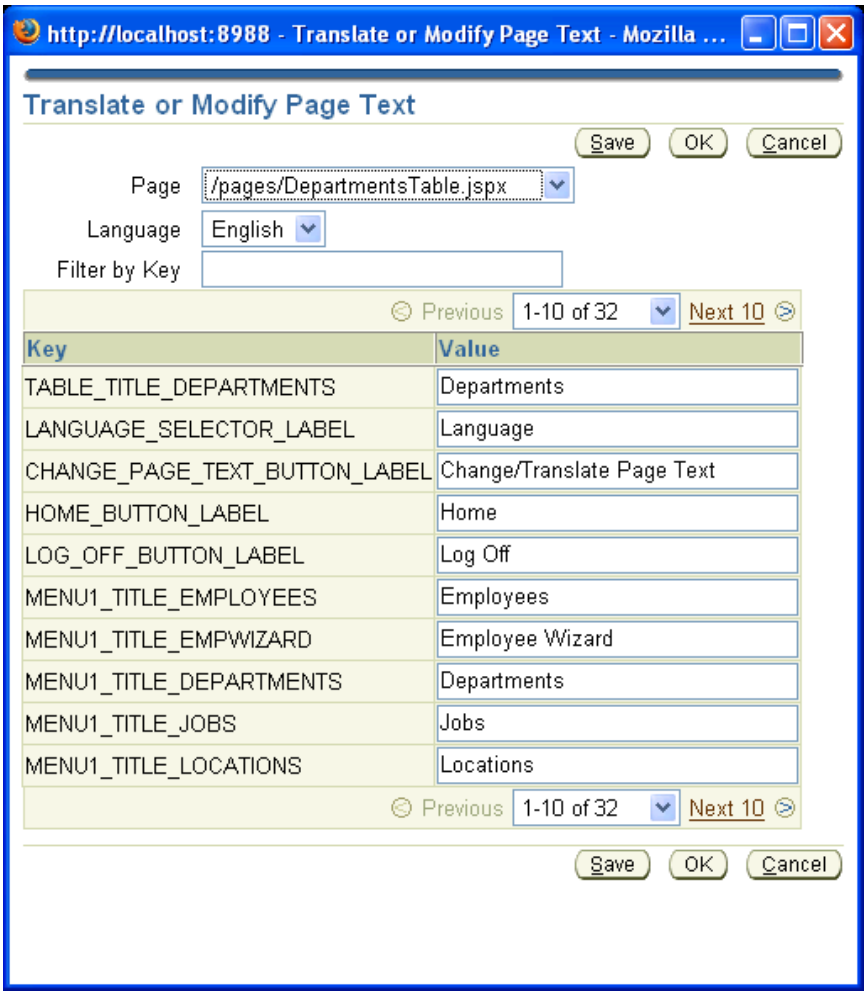


11.2.3. Running the Application

If you now run the generated application, you should see the Record Page Text button in the global button area.

If you enabled role-based authorization, the Record Page Text button is only visible when the logged in user has the ADMIN role, but you can easily change this by making a custom template for menuGlobal.vm.

When you now click this button, the JHeadstart runtime will "record" all translatable text strings in each page, and the Record Page Text button will be replaced with a button that can be used to invoke the ChangePageText dialog page.



In this dialog, you can select a page to translate/modify from a drop down list. This drop down list shows a list of all pages you visited after clicking on the Record Page Text button. Through the language drop down list, you can select the language for which you want enter/modify page translations.

11.3. Runtime Implementation of National Language Support

If you want to access a resource bundle in a JSF JSP page, you normally add a loadBundle tag to your page like this:

```
<f:loadBundle basename="oracle.srdemo.view.resources.UIResources"
var="res"/>
```

And then you can access entries in this resource bundle like this:

```
<af:panelPage title="#{res['srcreate.pageTitle']}">
```

While this is a simple technique, the drawback is that you explicitly have to name your resource bundles in your page, and if you have multiple resource bundles, you need to include multiple `<f:loadBundle>` tags, and you need to know which entry resides in which bundle.

JHeadstart takes a slightly different approach. Instead of generating `<f:loadBundle>` tags into the pages, JHeadstart generates a managed bean definition under the key `nls` which instantiates a class that provides access to all resource bundles of your application.

In generated pages, you will often see references to this `nls` managed bean like this:

```
<af:panelPage title="#{nls['TABLE_TITLE_EMPLOYEES']}">
```

This approach provides you with the flexibility to (re-)organize your resource bundles as you like, without the need to change the references to resource bundle entries in your page.

In addition, this approach 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 your application 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 make this all work, the following managed bean definitions are generated into the JhsCommon-beans.xml:

```
- <managed-bean>
-   <managed-bean-name>jhsMessageFactory</managed-bean-name>
    <managed-bean-class>oracle.jheadstart.controller.jsf.util.MessageFactory
    </managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
    <managed-property>
      <property-name>bundleNames</property-name>
      <list-entries>
        <value>view.ApplicationResources</value>
        <value>oracle.jheadstart.exception.JhsUserMessages</value>
        <value>javax.faces.Messages</value>
      </list-entries>
    </managed-property>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>nls</managed-bean-name>
    <managed-bean-class>oracle.jheadstart.controller.jsf.util.MessageFactoryMap
    </managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
    <managed-property>
      <property-name>messageFactory</property-name>
      <value>#{jhsMessageFactory}</value>
    </managed-property>
  </managed-bean>
```

The `MessageFactory` class is the class that loads all bundles specified through the `bundleNames` managed property. The `MessageFactoryMap` class is just a wrapper around the `MessageFactory` class that implements the `Map` interface, so we can use JSF EL expressions in the page to get entries from the resource bundle.

Reference: See the Javadoc or source of `MessageFactory` and `MessageFactoryMap`.

11.4. Error Reporting

The ADF SR Demo (ADF BC) by Steve Muench customizes the default way a "tree" of bundled ADF exceptions gets translated into JSF error message objects for display in the JSP page by overriding the `reportErrors()` method of the ADF `FacesPageLifecycle`.

If multiple exceptions are reported for the same attribute, error reporting is simplified 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: You can easily install the ADF SR Demo using the JDeveloper Check for Updates facility available under the Help menu. Note that you do need an internet connection when using Check for Updates.

JHeadstart has copied the SR Demo error reporting extension to the JHeadstart `ErrorReportingUtils` class and made some further customizations. This class is instantiated by looking up a managed bean definition in JhsCommon-beans.xml:

```
<managed-bean>
  <managed-bean-name>errorReportingUtils</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.util.ErrorReportingUtils
  </managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <managed-property>
    <property-name>expectedExceptionClasses</property-name>
    <list-entries>
      <value>oracle.jbo.AttrSetValException</value>
      <value>oracle.jbo.AttrValException</value>
      <value>oracle.jbo.JboException</value>
      <value>oracle.jbo.TooManyObjectsException</value>
      <value>oracle.jbo.RowValException</value>
      <value>oracle.jbo.TxnValException</value>
      <value>oracle.jbo.ValidationException</value>
      <value>oracle.jheadstart.exception.JhsFormattedJboException</value>
      <value>oracle.jheadstart.exception.JhsJboException</value>
    </list-entries>
  </managed-property>
  <managed-property>
    <property-name>printStackTraceUnexpectedException</property-name>
    <value>true</value>
  </managed-property>
  <managed-property>
    <property-name>showJBOErrorCode</property-name>
    <value>true</value>
  </managed-property>
</managed-bean>
```

Again, this approach brings you the advantage of easily configuring error reporting to meet your specific requirements. By using managed properties, you can configure

- whether the stack trace of unexpected exceptions is printed. By default it is printed.

- which exceptions should be treated as expected exception, for which the stack trace will never be written to the log.
- whether the error code (product code and error number) should be displayed.

Attention: To avoid losing your changes in the managed property values after generating again, you should create a custom template based on the JhsCommonBeans.vm template and make the changes in this custom template. To use your custom template rather than the default template, select the Service node in the Application Definition Editor, go to the Templates tab, and set the value of template key JHS_COMMON_BEANS to the name of your custom template. See chapter 4, section "Using Generator templates" for more information on using custom templates.

Note that using the same technique of creating a custom template, you can use your own subclass of ErrorReportingUtils when the exposed managed properties are not sufficient to meet your specific needs.

Additional functionality implemented in this class relates to database errors. 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 JSF Message stack, so you can provide a user-friendly message to the user by adding this constraint name as key to your message resource bundle. Note that the JHeadstart Application Generator already generates such messages into your message resource bundle for all key constraints defined in the XML file of your Entity Objects.

Reference: See the Javadoc or source of `ErrorReportingUtils.reportErrors`

11.5. 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 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.

Looking at a generated commandMenuItem will clarify how this works:

```
<af:commandMenuItem text="Employees" onclick="return alertForChanges();"
                    action="StartEmployees" immediate="true"
                    selected="#{attrs.selectedTab=='Employees'}">
  <f:actionListener type="oracle.jheadstart.controller.jsf.listener.DoRollbackActionListener"/>
  <f:actionListener type="oracle.jheadstart.controller.jsf.listener.ResetBreadcrumbStackActionListener"/>
  <af:resetActionListener/>
</af:commandMenuItem>
```

The JavaScript alert is shown because the onclick property calls function `alertForChanges()`in `form1013.js` (the JHeadstart JavaScript library). 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 item, 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 `form1013.js`. You can find `form1013.js` by going to your ViewController project, and opening `[HTML Root Directory]/jheadstart/form1013.js`.

CHAPTER

12

Runtime Page Customizations

When you deliver your application to multiple customers or organization units, these customers or organization units might have specific requirements for customizing the application. A typical example is an independent software vendor (ISV) who delivers an application to multiple customers. Each customer has specific requirements, for example a customer wants to add additional items to some pages, or they want to hide standard items. These requirements could be implemented by creating separate code bases for each customer, but this easily creates a maintenance nightmare.

JHeadstart offers extensive capabilities for runtime customization of pages, which allows you to support customer-specific requirements without changing the code base, and without changing the underlying database model.

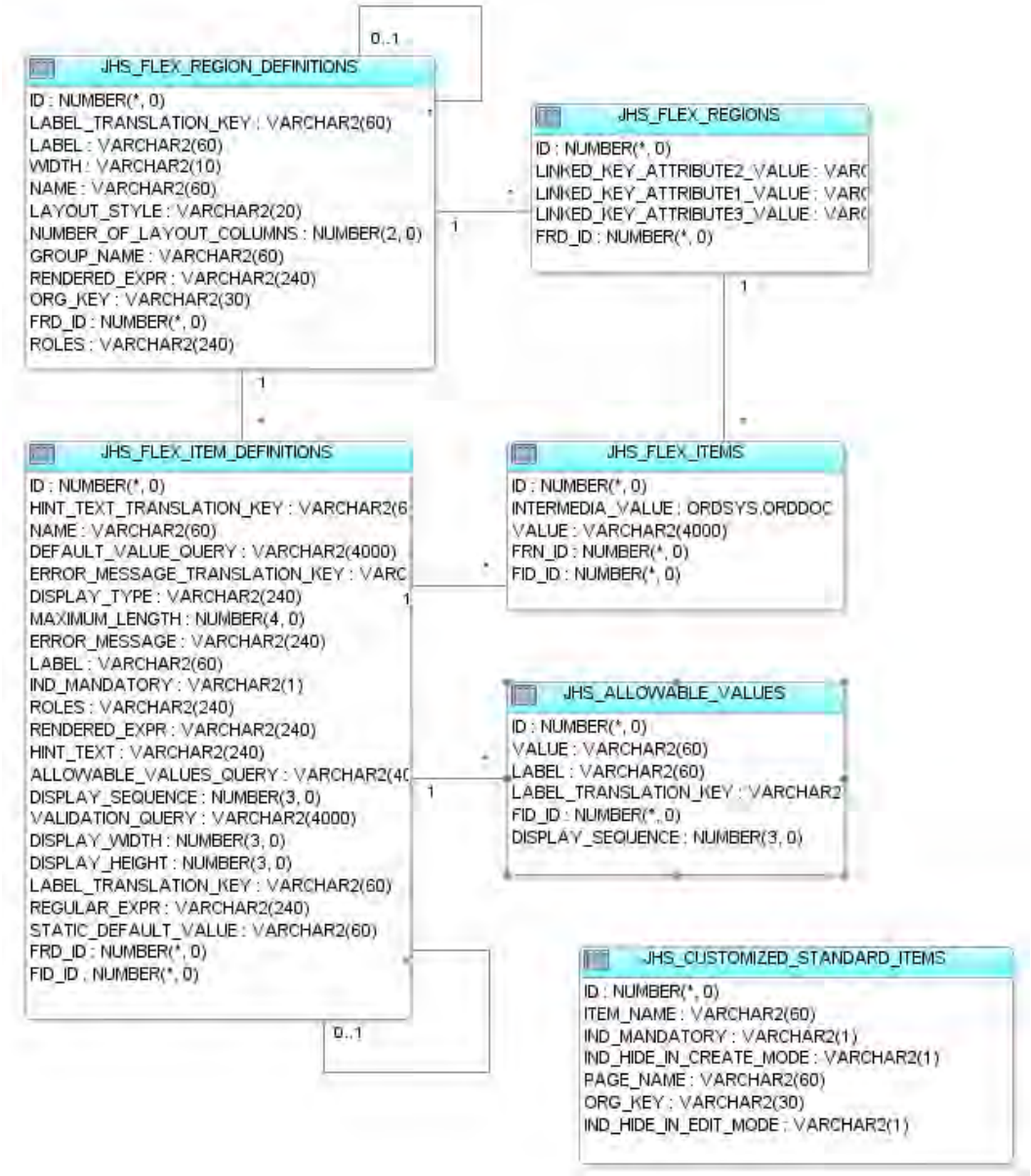
This runtime customization functionality can be split into two areas:

- Defining additional so-called Flex Items at runtime.
- Customizing standard items at runtime. It is possible to hide an item in create and/or edit mode, or to make an optional item required.

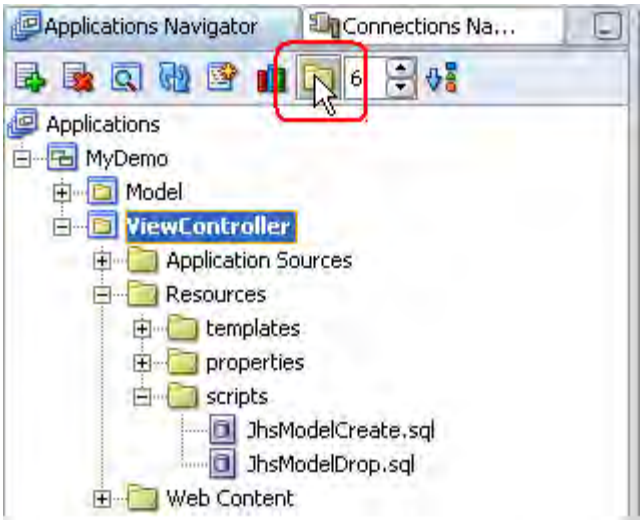
In the next sections we will explain how you can enable your application to use flex items and customized standard items.

12.1. Creating the Database Tables

JHeadstart uses a set of database tables to support these runtime customizations. The structure of these tables is shown below.



Before you can start using Flex Items or Customized Standard Items, you need to create the above table structure in your own application database schema. You can do this by running the script JhsModelCreate.sql against the database connection of your application schema. This script is located in the scripts directory of your ViewController project. If you don't see the scripts directory, make sure you click the Toggle Directories in the toolbar of the Application Navigator.



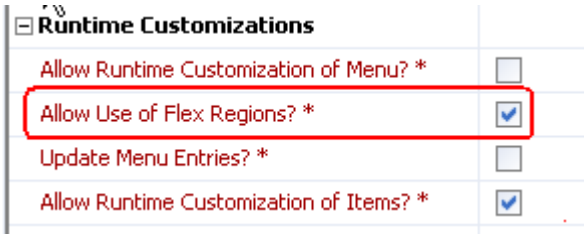
You can right-mouse-click on the JhsModelCreate.sql, then choose Run in SQL*Plus, and then the database connection you want to run the script in.

Attention: We recommend installing the JHeadstart tables in the same schema as your own application tables. If you nevertheless prefer to install the JHeadstart tables in a different database schema, then you need to ensure that your application schema has full access to the JHeadstart tables and synonyms with the same name as the table name. This is required because the JHeadstart runtime accesses the database tables through View Object usages defined in application module **JhsModelService**. When generating your application while using one or more of the table-driven features, this JhsModelService application module is added as a nested usage to your own application module, thereby "inheriting" the database connection of its parent application module.

Attention: The JhsModelCreate.sql script creates database tables for all table-driven JHeadstart runtime features. Additional tables for dynamic menus, translations and security are also created. If you do not plan to use these other features you can create your own script that only creates the above tables, and the JHS_SEQ sequence that is used to populate the ID column in these tables.

12.2. Enabling Runtime Usage of Flex Items

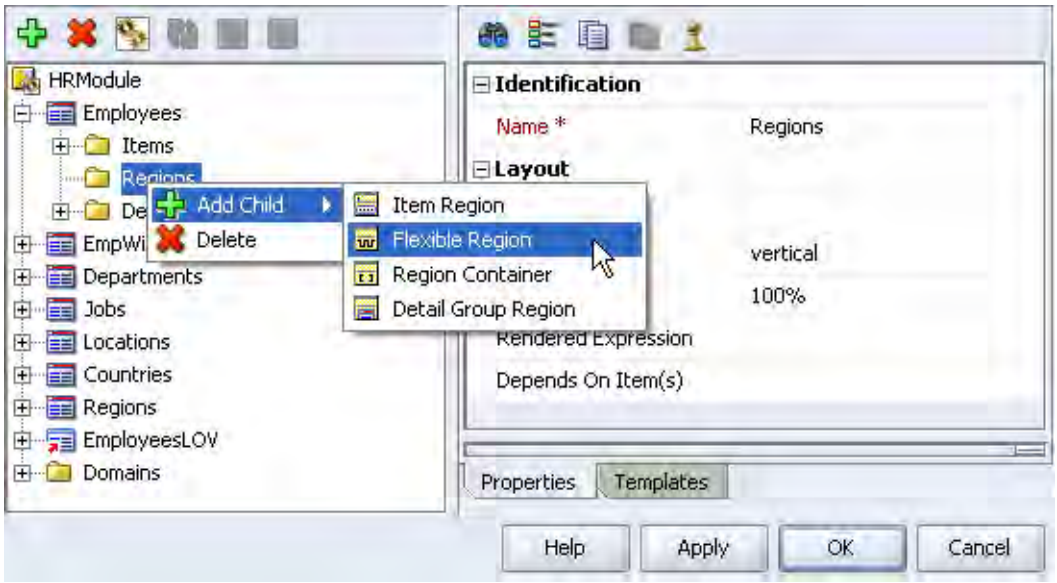
To enable your application for use of Flex Items, the first thing to do, is to check the service-level checkbox "Allow Use of Flex Regions".



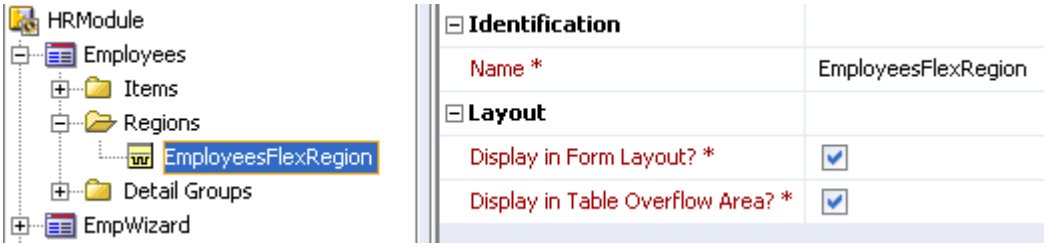
Next, you define "place holders" in the Application Definition Editor for a region of flex items that might be defined at runtime. A flex region placeholder can be defined in two ways: by creating a Flexible Region, or by creating an item with display type "flexRegion".

12.2.1. Creating a Flexible Region

To create a **Flexible Region**, you can right-mouse-click on the Regions icon within a group, and choose Add Child => Flexible Region.



Apart from the **Name**, a Flexible Region has two properties: **Display in Form Layout?** and **Display in Table Overflow Area?** . The first property is only applicable when the group has a layout style that includes a form page (form, table-form, select-form, tree-form). The second property is applicable when the group has a layout style that includes a table page (table, table-form) and the **Table Overflow Style** property is set on the group.

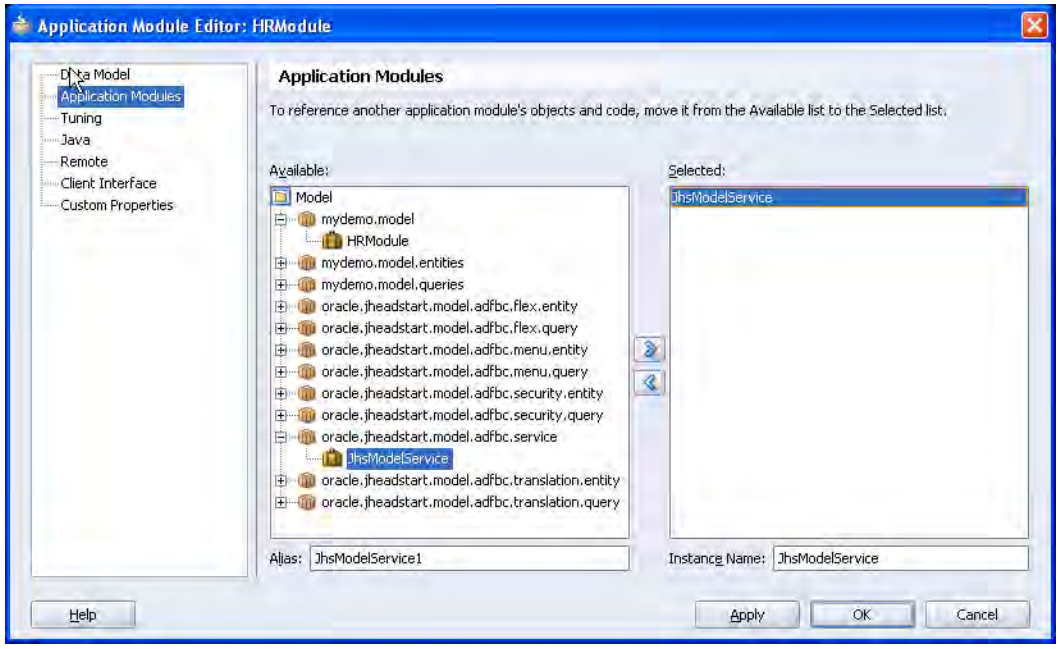


Note that you can define multiple Flex Regions for one group. For example, if you generate a wizard-style layout for your group, you might want to add a Flex Region to every wizard page.

12.2.2. Running the JHeadstart Application Generator

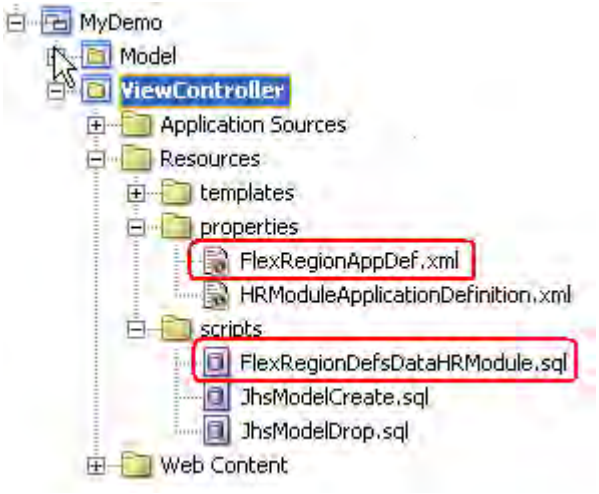
When you now run the JHeadstart Application Generator again, the following happens to enable flex items at runtime:

- All ADF Business Components included in the JHeadstart Runtime library are imported into your Model Project, and the **JhsModelService** application module, is added as a nested usage to your own application module. The **JhsModelService** includes View Object Usages that insert, update, delete and query the underlying database tables needed for the runtime customizations. Note that by creating JhsModelService as a nested application module, it will inherit the database connection of the parent application module, allowing for normal application data and flex region data to be committed in the same transaction.



- An additional Application Definition file, named FlexRegionAppDef.xml is generated. Click Save All after running the JAG and this file should appear into the properties directory of your ViewController project. You can use this Application Definition to generate the pages that are used to define the content of the Flex Region at runtime. See section "Generating the Flex Region Admin Pages" for more info.

- A SQL Script named FlexRegionDefsData*ServiceName*.sql is generated and executed against the default database connection of your ADF Business Components project. This script inserts rows in the JHS_FLEX_REGION_DEFINITIONS table for each Flex Region defined in the Application Definition. Note that if you do not want the JAG to auto-execute the script, you can uncheck the **service-level** checkbox "**Run Generated SQL Scripts?"**

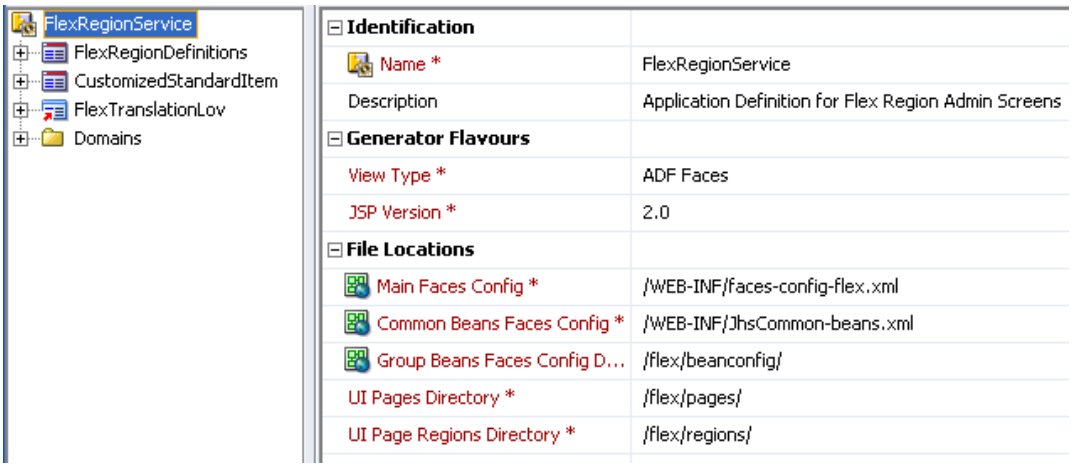


- The pages in a group with a Flex Region include ADF Faces elements that dynamically display the flex items that might be defined at runtime for this region. Of course, when you run the page just after you added the Flex Region, no flex items have been defined yet for this Flex Region and the page will look the same as it did without the flex items enabled.
- An additional global button "Customize Mode" is generated into the page. Clicking this button will allow you to invoke the flex region admin pages you are about to generate.

12.2.3. Generating the Flex Region Admin Pages

As explained above, a separate application definition FlexRegionAppDef.xml is generated that can be used to generate the Flex Region administration screens. This application definition is only generated when it does not exist yet, so after it has been generated, you can make any changes you want using the Application Definition Editor, without loosing these changes when you regenerate your "own" application definition.

Before you generate the FlexRegionAppDef you might want to inspect the file locations properties, and change them if you have set other naming standards for these properties.



The default settings are in line with what we recommend:

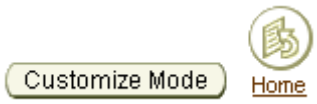
- Generate all Main faces-config files in the WEB-INF directory with a suffix indicating the service
- JhsCommon-beans.xml is shared by all services (application definitions); the location should never be changed.
- Create a subdirectory per service for the generation of group beans faces config files, pages and regions. Make a subdirectory under this "service" directory for each of these file types.
- The **View Package** property should be the same across all services
- The **Resource Bundle Type** and **NLS Resource Bundle** properties should be the same across all services.

Note that all service-level properties that are generally the same across all services have got the same values as in your "own" application definition. If you later on decide to make changes to these settings in your own application definition, then you will need to make the same change in the FlexRegionAppDef Application Definition.

Now, if you are satisfied with the settings, you can run the JAG for the FlexRegionAppDef.

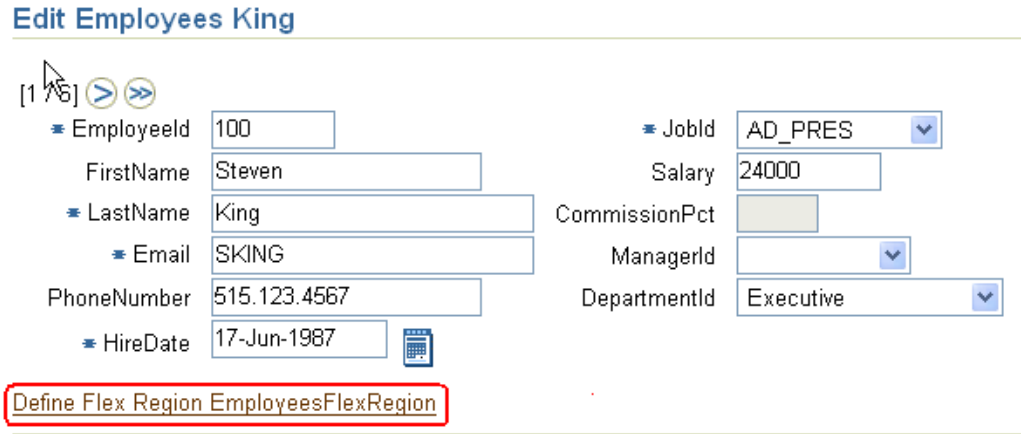
12.3. Defining Flex Items At Runtime

You are now ready to run the generated application, and define some flex items at runtime. If you start the application again, you will notice an additional button with label "**Customize Mode**" in the upper right corner of each page.

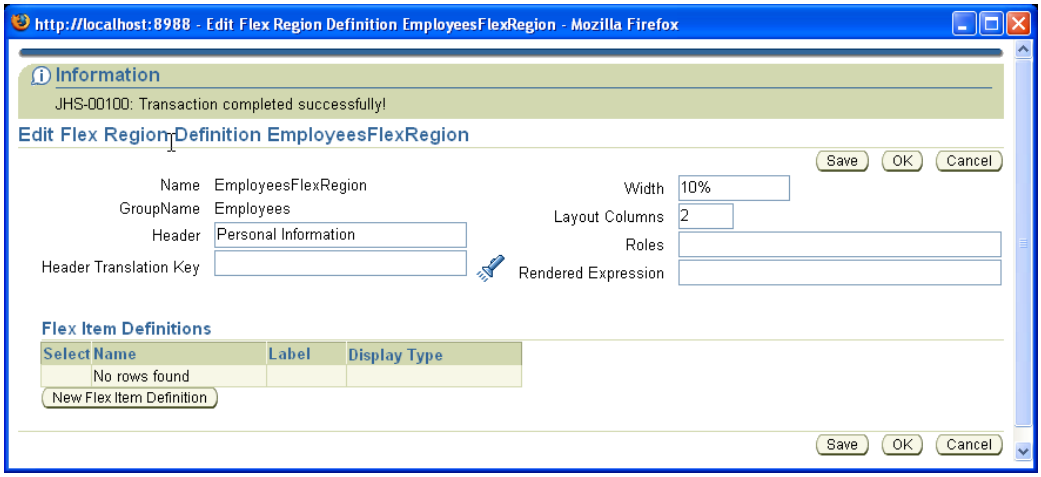


Note that if you have enabled role-based security in your application, you will only see the button when you log in as a user with a role as specified in the service-level property **Admin Role**. You can customize the appearance of the Customize Mode button, as well as the role required to see the button by modifying the menuGlobal.vm template.

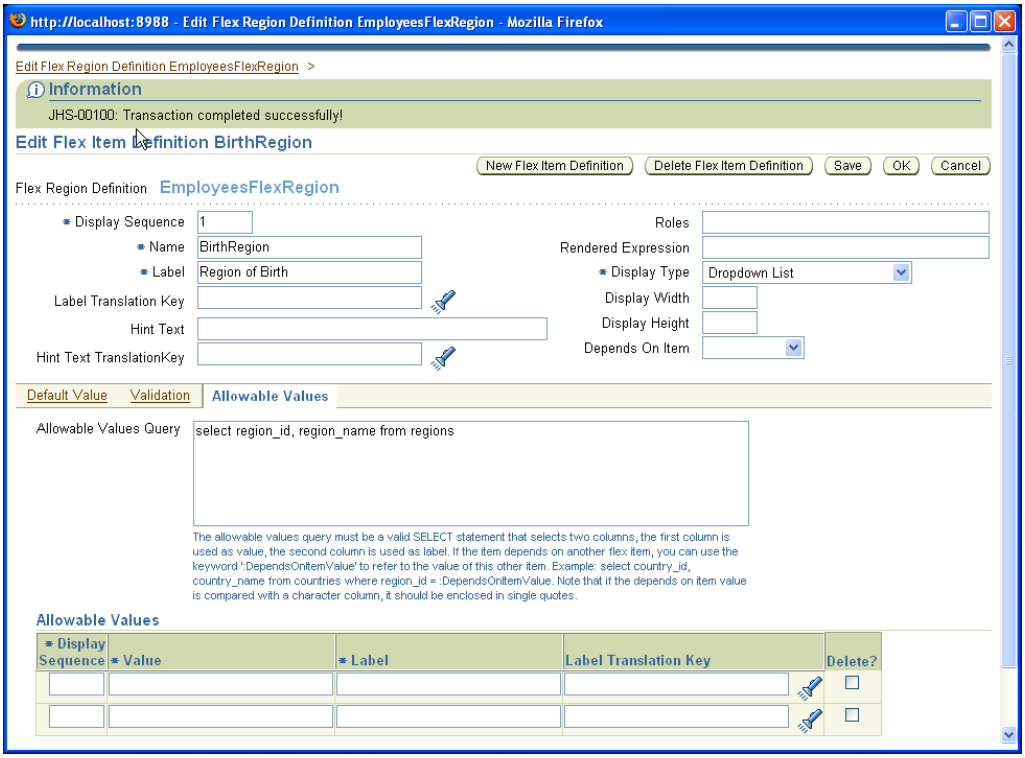
If you click the **Customize Mode** button on a page with a flex region defined, a link will appear at the location of the flex region.



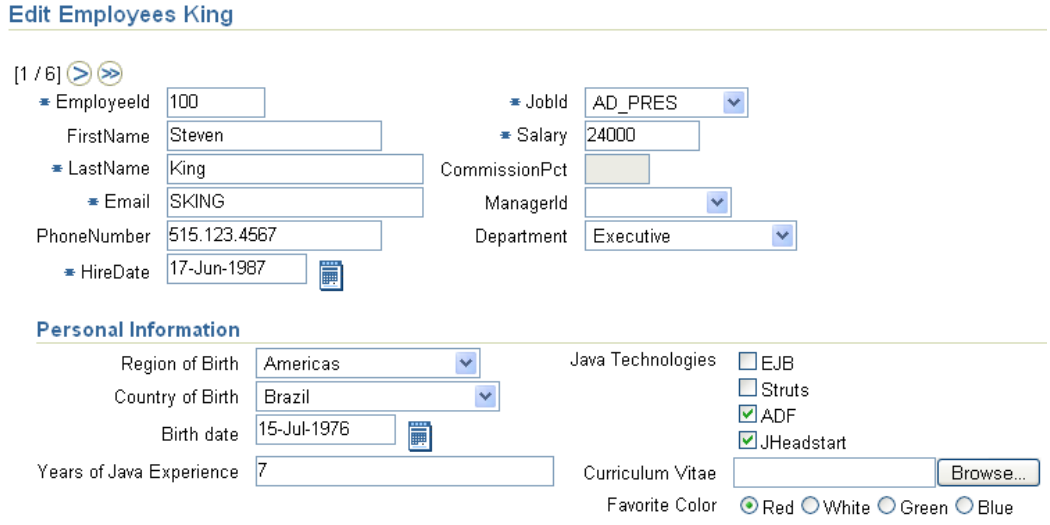
Clicking this link will launch a dialog window with pages where you can define the appearance of the Flex Region, and the Flex Items that should appear within the Flex Region. You just generated these dialog pages using the FlexRegionAppDef.



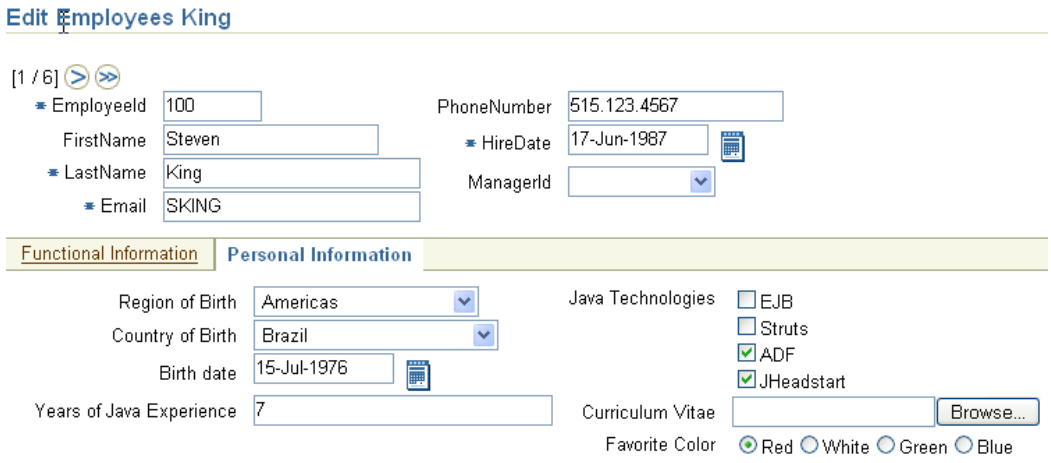
When you click the New Flex Item Definition button, you will get the following page in the dialog that allows you to define a flex item within the region.



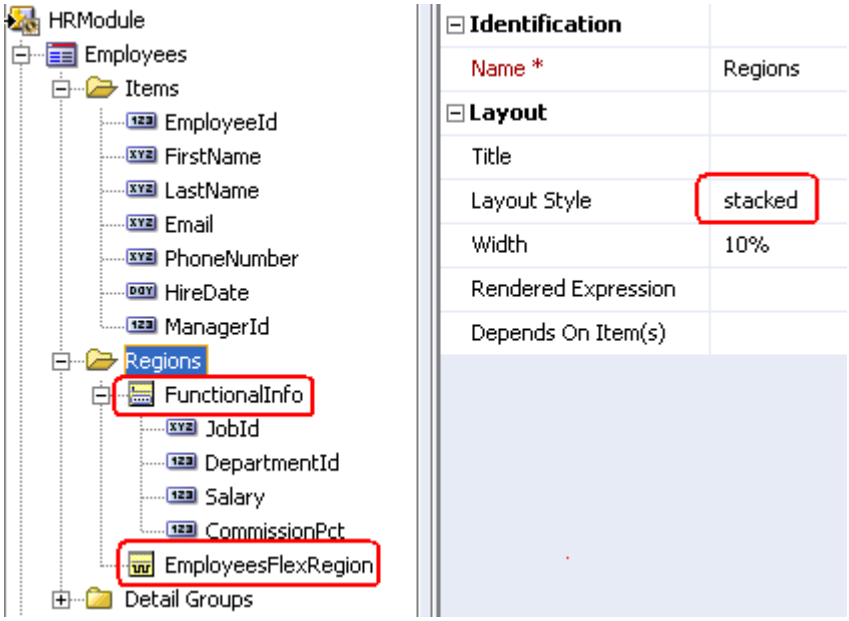
- In this page, you can define the display properties of the flex item, as well as default value logic, validation logic, and allowable values. When you are done defining the flex items, you can close the dialog. If you now navigate to another row in your page, you will see the flex items appear. To increase performance, the flex items are queried only once for each "base" row in a session, that's why you do not see the flex items for the employees your already visited in the same browser session prior to defining the flex item.



In this screen shot, the flex region is displayed below the normal items. You can also "stack" the flex region as shown below.



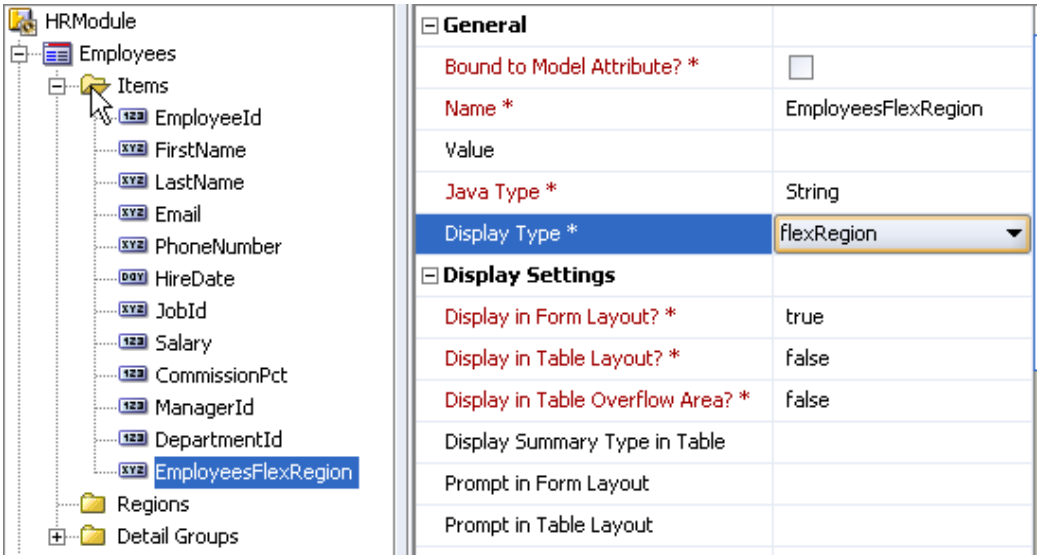
Setting up another Item Region and setting the Layout Style of the Regions container to "stacked" accomplish this.



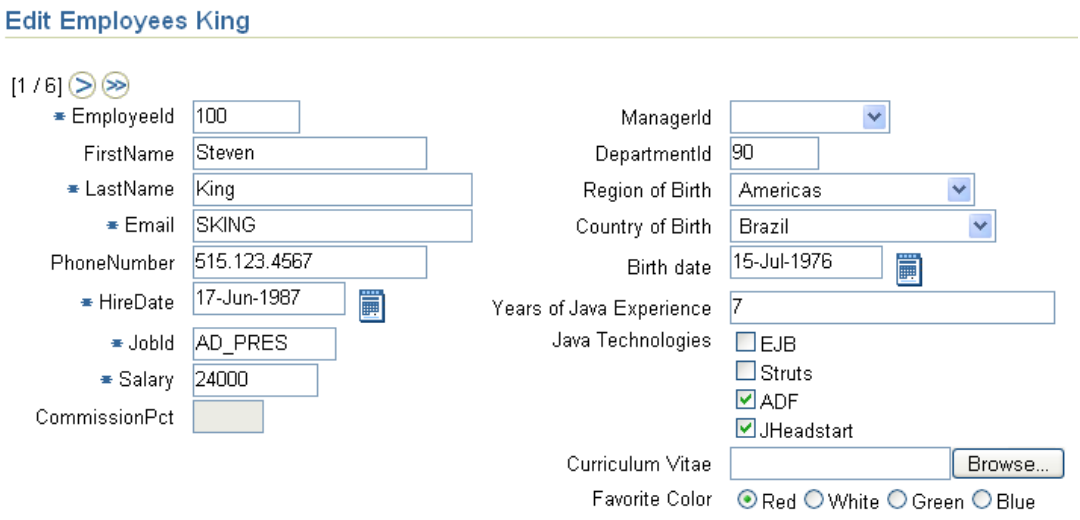
12.4. Creating an Item with Display Type Flex Region

By defining a Flex Region in the Application Definition, the flex items will always be displayed in their own visual region. You can control how this region is displayed relative to the standard items as we have seen above.

Now, if you want to display flex items within the same visual region as standard items, you can use a special item with **Display Type** "flexRegion" as the place holder for the flex items defined at runtime. Uncheck the checkbox Bound to Model Attribute for this item, and set Display In Table Layout to false, since flex items cannot be displayed in a table, only in a table overflow area.



If we now generate again and run the application, the flex items appear seamlessly with the standard items, as shown in the screen shot below.



Note that you can define an unlimited number of **Flex Regions** and Items with **Display Type** "flexRegion" in a group.

12.5. Internationalization and Flex Items

Flex regions, and the flex items within a flex region support multiple languages. In the dialog pages you use to define the flex region and flex items, you might have noticed that every text item has a corresponding **Translation Key** item.

When you enter a value in a **Translation Key** item, this value will be stored as translation key in the JHS_TRANSLATIONS table. The value of the corresponding text item will be stored as translation text. For each **Locale** specified in the Application Definition, an entry will be created.

JHeadstart must be configured to use the JHS_TRANSLATIONS table to retrieve translatable strings. This is done through the service-level property **NLS Resource Bundle Type**, which must be set to "databaseTable". If for whatever reason you do not want to use the JHS_TRANSLATIONS table as your resource bundle, then you should not enter a value in the **Translation Key** items. In this case, we recommend you change the FlexRegionAppDef application definition, and set the **Display in Form Layout?** Property of all Translation Key items to "false".

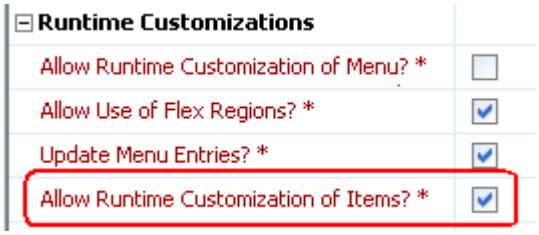
Please refer to the Internationalization section of this chapter for more information on resource bundle types, and the option to change and translate your pages at runtime.

12.6. Customizing Standard Items at Runtime

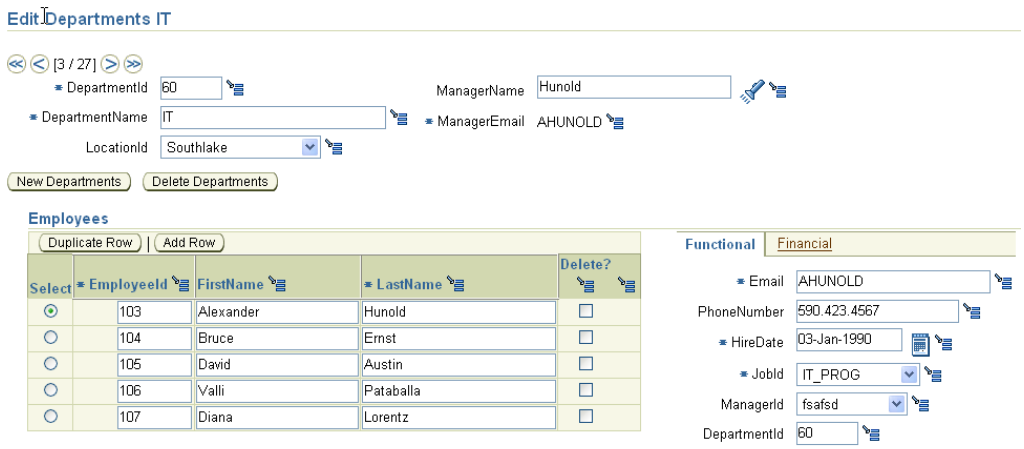
Standard generated items can be customized at runtime to a certain extent:

- Items that are generated as optional can be made required
- Items can be completely hidden, or conditionally hidden based on whether the user is creating a new row or editing/viewing an existing row.

To enable customization of standard items at runtime, you need to check the service-level checkbox "**Allow Runtime Customization of Items?"**



After you generated the application and you run the application again, the "Customize Mode" button will appear. If you already enabled flex items in your application, the Customize Mode button was already there. If you click the "Customize Mode" button after you generated with the above setting, all items and column headers have a customize icon displayed at the right of the item.



When you click the customize icon, it launches a dialog in which you can customize the icon for which you clicked the icon.

After you saved your changes and closed the dialog, the changes will not be visible until you clicked the "Normal Mode" button, and you navigated to another page. When you then return to your customized page, the runtime customization is applied.

CHAPTER

13 Forms2ADF Generator

The JHeadstart Forms2ADF Generator (JFG) allows you to reuse Oracle Forms elements and properties when creating Oracle ADF applications.

The JFG creates the Business Services (ADF Business Components) and the JHeadstart meta data (Application Definition). After that you can run the JHeadstart Application Generator to generate an ADF web application based on the User Interface definitions that have been extracted from the Oracle Form.

This chapter explains how the JFG works, and how to use it.

13.1. Introduction into JHeadstart Forms2ADF Generator (JFG)

If you have used Oracle Forms as your development environment over the years, the JHeadstart Forms2ADF Generator (JFG) can be used in different situations to move to Oracle ADF. Typical scenarios where the JFG can be of use include:

- You want to add self-service functionality to existing Oracle Forms back-office applications
- You want to leverage advanced user interfaces features in JDeveloper/ADF Faces which are hard or impossible to build in Oracle Forms
- You want to disseminate Oracle Forms artifacts in J2EE to prepare for a transition to a service-oriented architecture (SOA).
- You want to migrate (parts of your) Oracle Forms applications to ADF.

While the JFG can assist in migrating Oracle Forms to ADF, it is not a migrator in itself. Any custom PL/SQL logic residing in the form will not be migrated to ADF. See section "Handling Forms PL/SQL Logic" for more information on Forms PL/SQL logic.

The Forms2ADF generation process looks as follows:

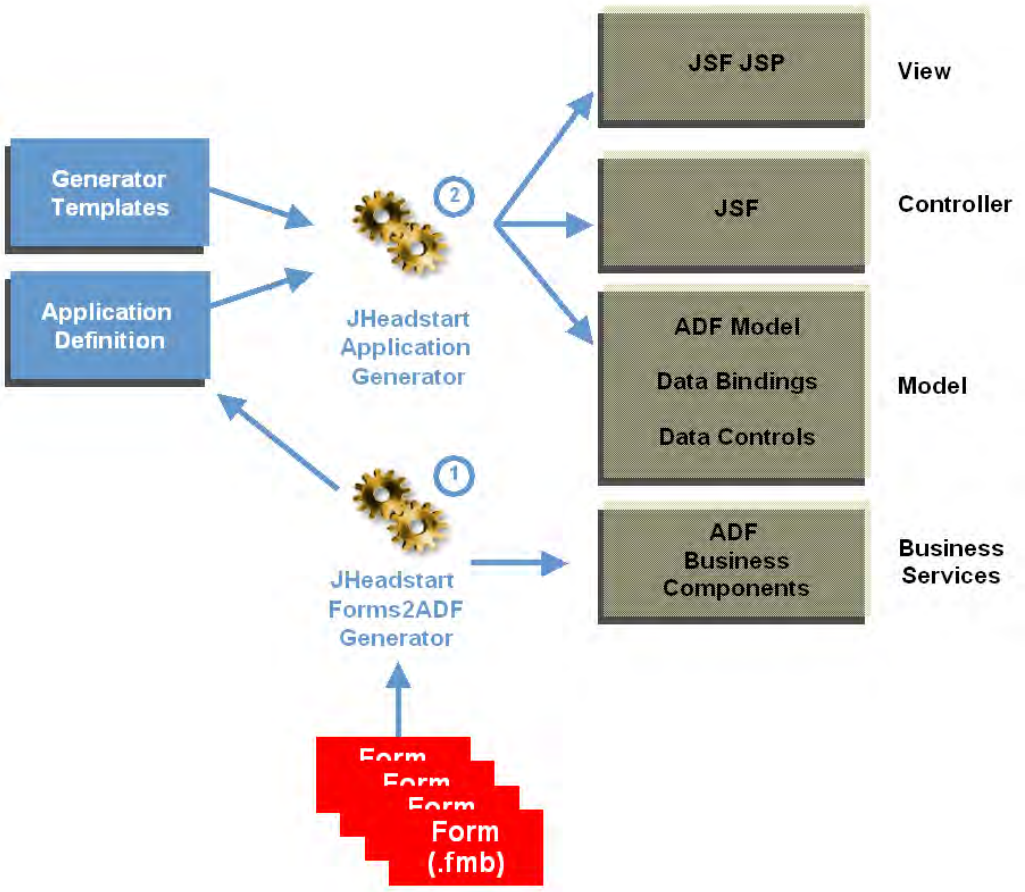


Figure 13-1 JHeadstart Forms2ADF Generator Process

The JHeadstart Forms2ADF Generator creates two main outputs:

1. ADF Business Components based on the data usages in the Oracle Form:
   - ADF BC Entity Objects are created for each table used by a Forms block.
   - ADF BC View Objects are created for each Forms data block and each record group query. Named query Bind parameters are created based on references to :block.item in the query WHERE clause.
   - ADF BC Application Modules are created for each form. All form-based application modules are nested inside one overall root application module.
2. JHeadstart Application Definition based on the user interface definitions in the form:
   - Groups are created for each block.
   - LOV Groups are created for each LOV / Record Group
   - Group Items are created for each item in a block.
   - (Stacked) region containers and regions created based on item placement on (tabbed) canvasses and within framed graphics
   - Domains created based on forms item allowable values
   - The PL/SQL logic in the form is extracted and added as "documentation" nodes under the group and item elements.

For m ore detailed description of the mapping between Forms elements and ADF Business Components and the JHeadstart Application, see section *"Understanding the Outputs of the JHeadstart Forms2ADF Generator"*.

13.2. Roadmap

When you plan to use the JFG, it is recommended that you follow the steps below:

1. **Make sure Forms .fmb file is version 9i or 10i.**

   The JFG uses the Forms utility to convert a forms .fmb file to XML. This utility is available as of Forms version 9i. If you built your forms with an older version, you need to open your forms in version 9i or 10i, and save the .fmb file with this version. Note that you do NOT need to upgrade your whole forms application to 9i or 10i; the JFG only needs the .fmb file in the proper version.

2. **Analyze the forms for elements that should be ignored by the JFG**

   When running the JFG you can specify names of form elements that should be ignored during the generation process. To choose the appropriate elements to exclude from generation, you need to analyze the forms you want to run through the JFG. Typical candidates to exclude are:

   - Common forms elements added through an object library, like a block, canvas and window to display a calendar popup on date items, or a canvas and window to display errors.
   - Current record indicator items
   - Query-Find blocks, windows and canvasses

3. **Prepare your project in JDeveloper**

   Before you can run the JFG you must create and prepare your project in JDeveloper. See Chapter 1 '*Getting Started*' on how to prepare your JDeveloper project, and apply the steps until just before the creation of new ADF Business Components.

4. **Run the JFG**

   You are now ready to actually run the JHeadstart Forms2ADF Generator in your Model project. It is recommended that you start with the simplest forms in your application to build experience with the JFG and the results it produces. See the next section on how to start and use the JFG.

5. **Understand and Inspect the Forms2ADF Generator Outputs**

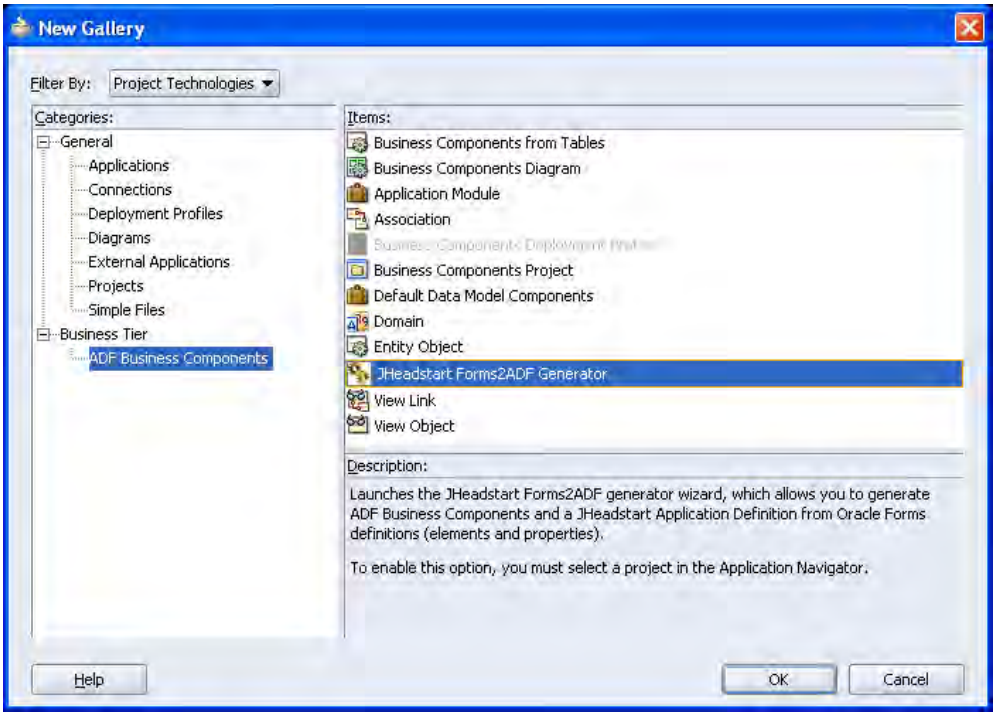
   Before generating the web application using the JHeadstart Application Generator, we recommend you check the outputs produced by the JFG. See section "*Understanding the Outputs of the JHeadstart Forms2ADF Generator*" for more information.

6. **Create ADF Web Application using the JHeadstart Application Generator**

   After running the JFG you can generate JHeadstart applications in the same way as if you would have created your ADF Business Components manually, and as if you would have run the New JHeadstart Application Definition wizard. See Chapter 1 '*Getting Started*', and apply the steps after creation of a JHeadstart Application Definition.

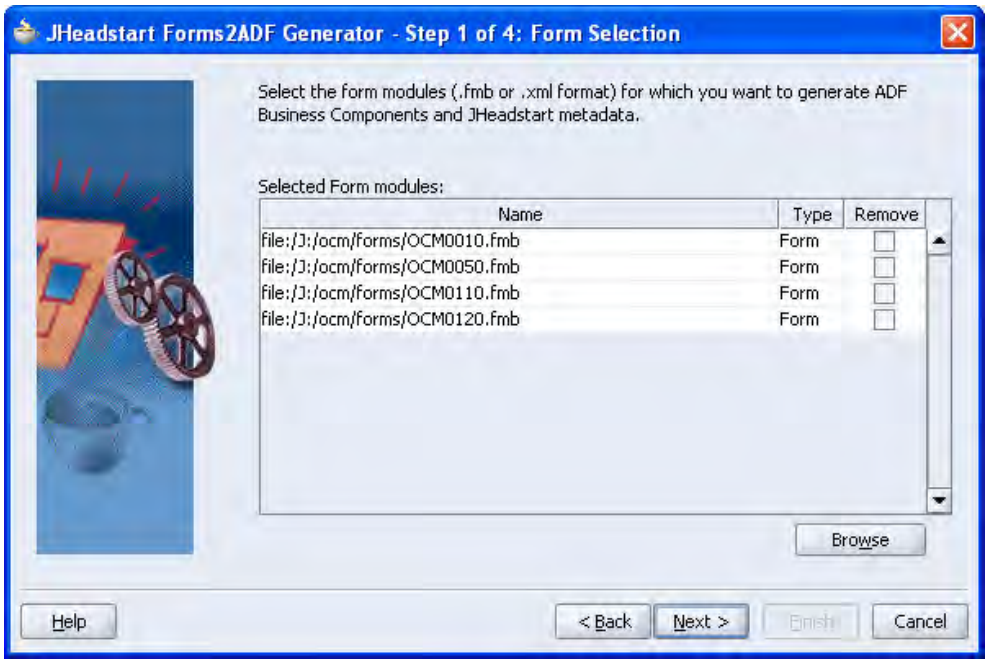
13.3. Running the JHeadstart Forms2ADF Generator (JFG)

You should start the JHeadstart Forms2ADF Generator from the Model project. To start the JHeadstart Forms2ADF Generator select your Model project in JDeveloper, right-mouse click, select New (or from the Menu, select File -> New), go to the Business Components node below the Business Tier. Select JHeadstart Forms2ADF Generator.



13.3.1. Select Forms Modules

You can select Oracle Forms .fmb files, or you can first use the Forms frmf2xml utility to convert the forms to xml format, and then select the converted xml files.



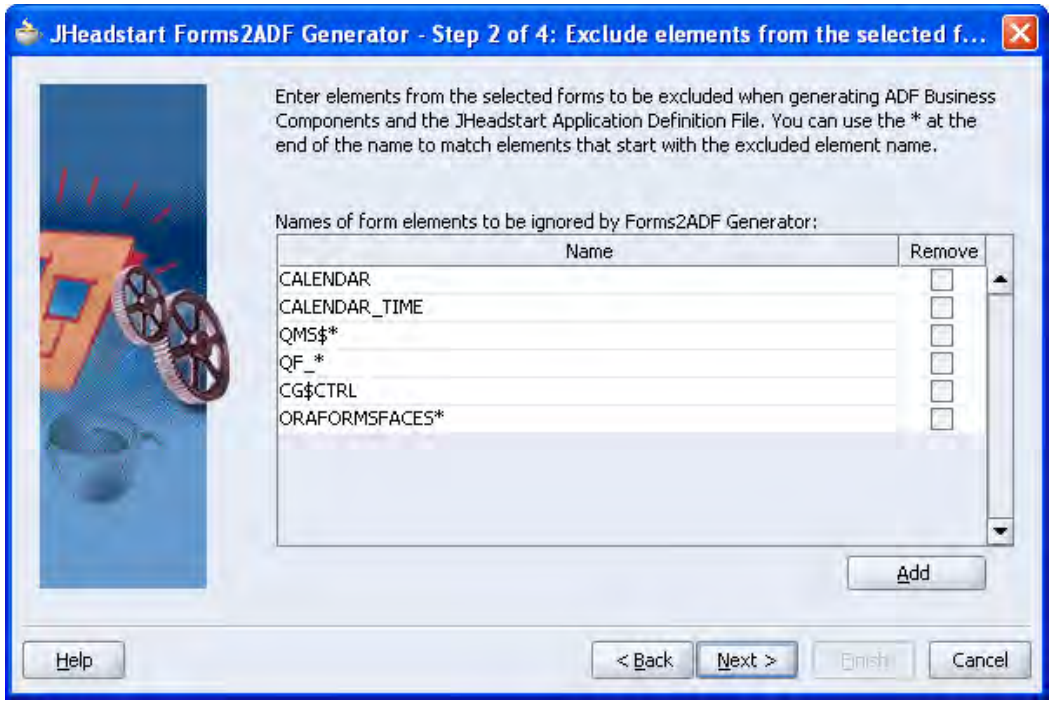
You can use the Browse button to open a File Chooser window to easily select the .fmb or .xml files from your file system.

If you select Forms .fmb files, then the JFG will first run the Forms frmf2xml utility under the covers. This utility requires that Oracle Forms be installed on the machine on which you run JDeveloper. If you do not have Oracle Forms installed, you can run the frmf2xml utility on another machine where Oracle Forms is installed, and then select the converted XML files in the File Chooser window.

If you nevertheless select an .fmb file while Oracle Forms is not installed, you will get an error message when you press the Next button.

13.3.2. Select Form Elements to be Excluded from Processing

Most of the Oracle Forms you want to process typically include elements that you want to ignore during processing by the JFG. In step 2 of the JFG you can define this list of elements you want to ignore. The match is based on the name, if you specify the name “CALENDAR” and your form contains a block, a window and a canvas all named “CALENDAR”, then all three elements will be ignored.



The default list of elements to exclude provides typical examples of such elements:

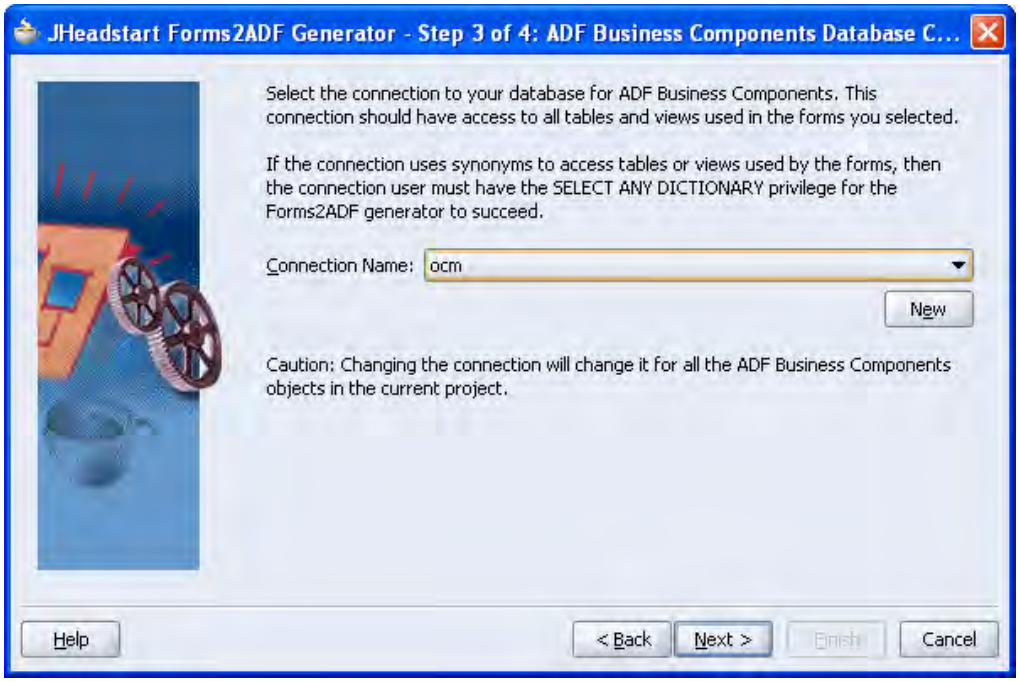
- CALENDAR, CALENDAR_TIME: ADF Faces provides its own built-in functionality for displaying calendar pop ups on date and date time fields
- QMS$*: Elements that start with this name exist in forms generated with the Template Package of Headstart for Oracle Designer. These are generic elements, like the current record indicator that are of no use in the ADF environment.
- QF_*: Elements that start with this name implement so-called Query Find functionality that can be generated with the Template Package of Headstart for Oracle Designer. Query Find windows do not need to be migrated, since JHeadstart has built-in support for quick and advanced search, similar to the Forms query find window.
- CG$CTRL: This block is added to the form when the form is generated using Oracle Designer. The block contains control items for Forms-specific logic that does not map to ADF concepts.
- ORAFORMSFACES*: A block and canvas of this name is included in forms that are enabled for inclusion in a JSF page using the OraFormsFaces component supplied by Commit Consulting.

Reference: For information about OraFormsFaces, visit the website of Commit Consulting: http://www.commit-consulting.com/oraformsfaces/

13.3.3. Select Database Connection

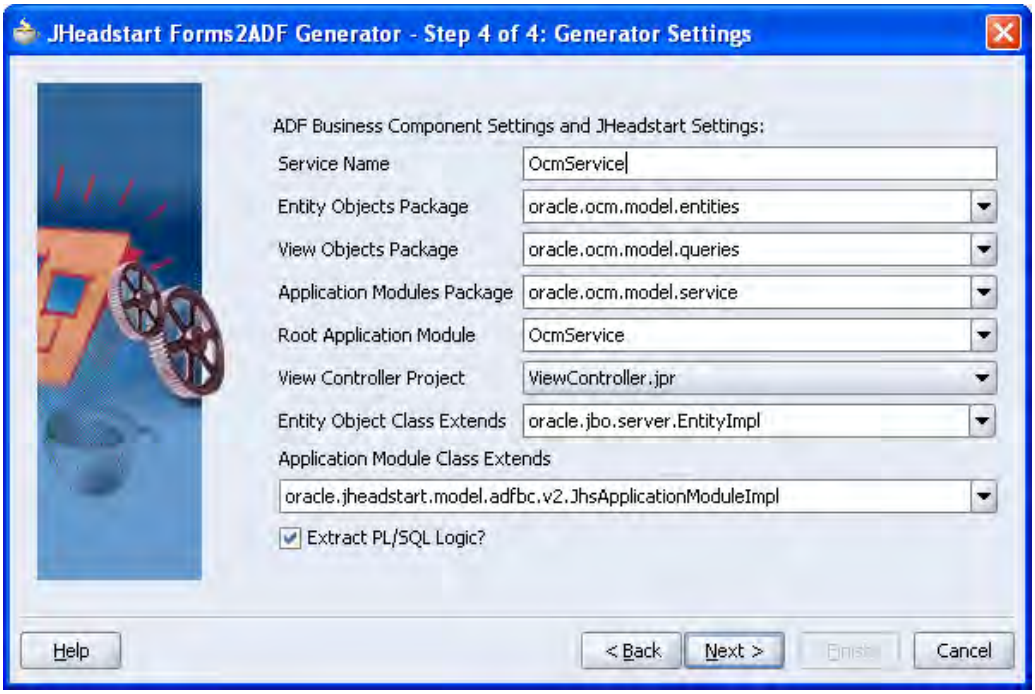
The database connection you select in this step is used as the connection for the ADF Business Components that will be generated. This connection is also used to query the table structures and key constraints from the Oracle Database Data Dictionary. This data dictionary information is required to create the ADF BC Entity Objects for each table used by a block in the selected forms. The ADF BC entity objects created by the JFG, are reused *across* all forms that are processed by the JFG, only the generated View Objects are

form-specific. Therefore, the entity objects created by the JFG contain attributes for all columns in the table or view, regardless of whether the column is used in one of the forms.



If the database schema you provide as database connection is not the *owner* of all tables and views used by the selected forms, then make sure that the schema has synonyms to these tables and views, and that the schema user is allowed to query the data dictionary tables for these tables and views. This can be accomplished by granting the SELECT ANY DICTIONARY privilege to the schema user. If your forms access tables or view in another database schema by prefixing the table or view with the schema name, rather then using synonyms, then the schema user you select should also have the SELECT ANY DICTIONARY privilege to be able to read the table and view definitions from this other schema.

13.3.4. Generator Settings



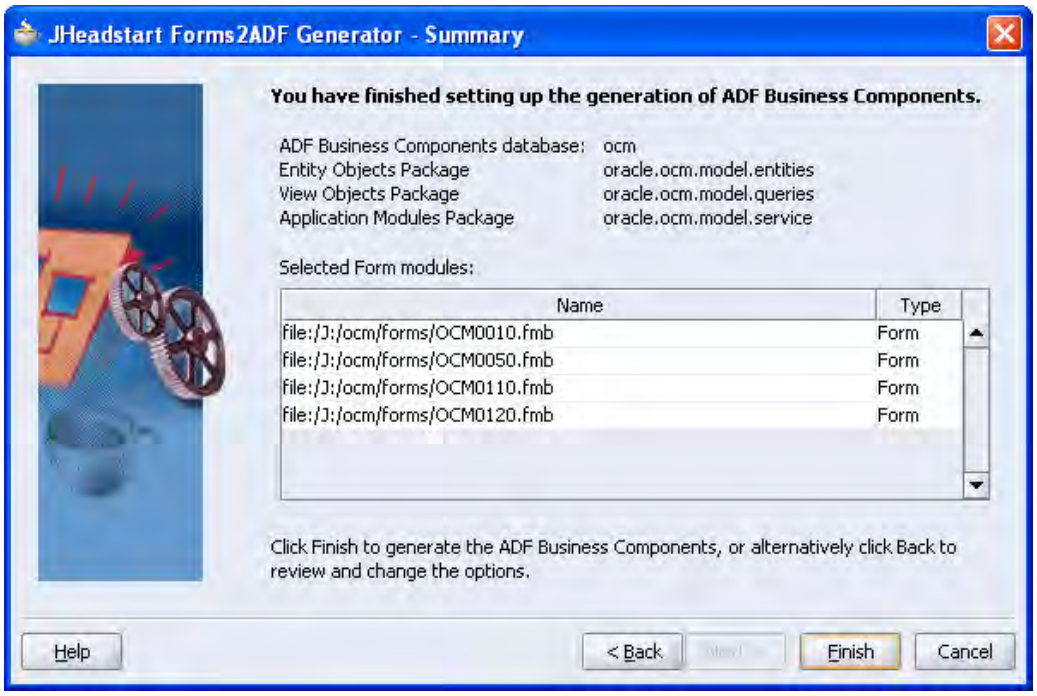
The following properties need to be set on the Generator Settings panel:

- **Service Name**: This is the name that is used to create the JHeadstart Application Definition file
- **Entity Objects Package**: The package name that will be used to store all entity objects and their associations. It is good practice to organize the entity objects, view objects and application modules in separate packages. If you specify a package name that does not yet exist, it will be created automatically.
- **View Objects Package**: The name of the package that will be used to store all view objects and view links.
- **Application Modules Package**: The name of the package that will be used to store all application modules.
- **Root Application Module**: The name of the root application module that will be created. For each form that you selected to process, a separate application module will be created, named after the forms module. All form-specific application modules are then nested inside one root application module, so they can share the same application module pool, and database connection pool at runtime.
- **View Controller Project**: The name of the project in which the JHeadstart Application Definition file will be saved.
- **Entity Object Class Extends**: The name of the superclass of each entity object that will be created by the JFG. You can create your own superclass that extends from the default `oracle.jbo.server.EntityImpl` class to implement common behavior across all your entity objects.

- **Application Module Class Extends**: The name of the superclass of each application module object that will be created by the JFG. You can create your own superclass that extends from `oracle.jheadstart.model.adfbc.v2.JhsApplicationModuleImpl` class to implement common behavior across all your application modules. If you use CDM RuleFrame to implement your business rules in the database, you should select the `oracle.jheadstart.model.adfbc.v2.RuleFrameApplicationModuleImpl` class, or your own subclass of this class. By extending from the `RuleFrameApplicationModuleImpl` class, your application will nicely display CDM RuleFrame errors in the web user interface.
- **Extract PL/SQL Logic?**: If this checkbox is checked, the PL/SQL program units, and the form-, block- and item-level triggers will be visible in the JHeadstart Application Definition. This is helpful in assessing the additional functionality that was implemented in the original form using PL/SQL that might need to be implemented as well in the web pages generated by the JFG/JAG.

13.3.5. Processing the Selected Forms

When you click the Finish button on the Summary panel, the JFG will start processing the forms.

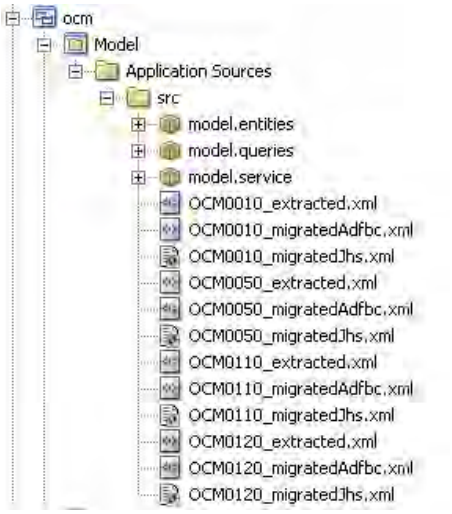


The processing consists of the following steps:

- **Conversion**: Each selected .fmb form is converted to XML format using the FRMF2XML utility. This step is skipped when you already selected xml-formatted form files.
- **Extraction**: For each data block in the form, the underlying table or view definition, including all primary keys, unique key and foreign key constraints are queried from the data dictionary tables. This information is added to the XML representation of the form, and the combined information is written to the file system, in the Java source root directory, in a file named <module_name>_extracted.xml

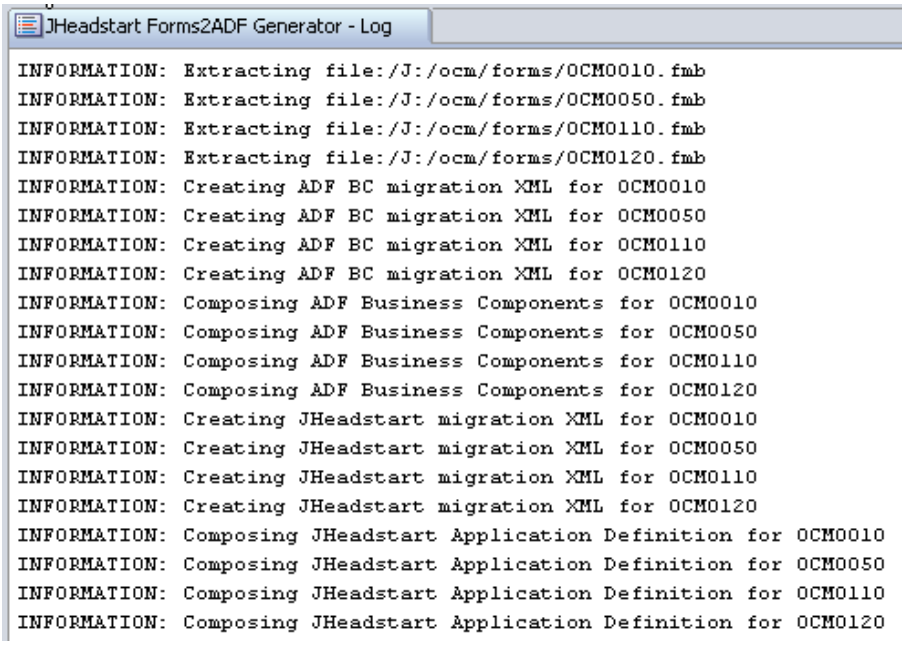
- **ADF BC Migration**: Each extracted XML file is processed by a number of ADF BC migrators (each target ADF BC element has its own migrator), that together create an XML structure that is the input for the actual creation (composition) of ADF BC components. This XML structure is written to the file system, in the Java source root directory, in a file named <module_name>_migratedAdfbc.xml
- **ADF BC Composition**: Based on the migrated ADF BC XML structure of each form, the entity objects, entity associations, view objects and view links, and the application modules are created.
- **JHeadstart Migration**: Each extracted XML file is then processed by a number of JHeadstart migrators (each target JHeadstart Application Definition element has its own migrator), that together create an XML structure that is the input for the actual creation (composition) of the JHeadstart Application Definition file. This XML structure is written to the file system, in the Java source root directory, in a file named <module_name>_migratedJhs.xml
- **JHeadstart Composition**: Based on the migrated JHeadstart XML structure of each form, the JHeadstart Application Definition file is created, and stored in the properties directory of the ViewController project.

If the ADF2Generator run was successful, you can safely delete the intermediate results of the Forms2AdDF Generator, being the extracted and migrated XML files in the Java source root directory. However, if processing failed with an error, these XML files can be used for troubleshooting as explained in the next section.



13.3.6. Troubleshooting

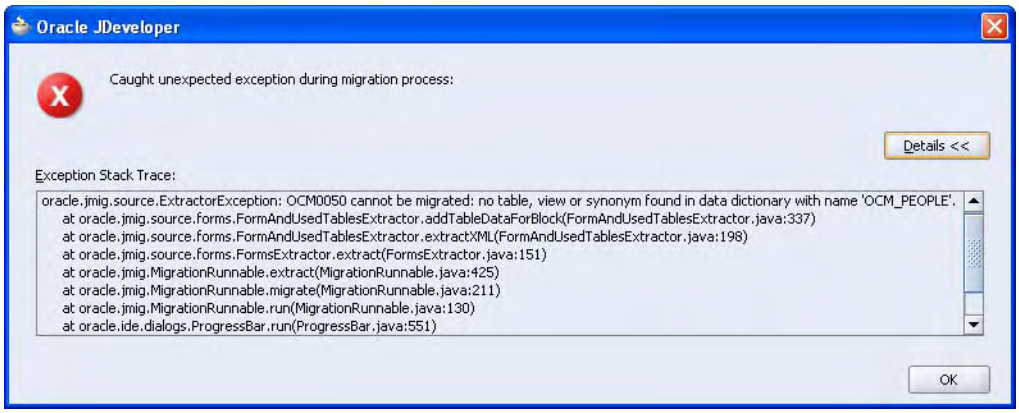
When the JHeadstart Forms2ADF Generator fails with an error, the first thing to do is to assess which form module is causing the error. This information can easily be obtained from the log window in JDeveloper, which prints an informational message for each processing phase, for each module, as show in the screen shot below.



The last printed line in this window will tell you which forms module the JFG was processing when the error occurred, and in which phase of the processing.

Typically, an unexpected error dialog window will be displayed, which provides more information about the error that occurred.

For example, the error dialog below will be displayed when no table or view information could be queried from the Oracle data dictionary tables during the extraction phase of a form module.



To solve this error, make sure that the database connection you specified has all the required privileges. See section 13.3.3 “Select Database Connection” for more information.

While you can fix the above error, most other unexpected errors might be related to a problem in the JFG code base. While the JFG has been tested against numerous simple to very complex forms, it is still in preview status. It has not been proven yet in a large-scale customer project. There might be (combinations) of form element definitions in your oracle forms that cannot be handled by the JFG. Or, the JFG will run and finish successfully, but the outputs do not match your expectations based on the form definition.

In such a situation, we recommend that you e-mail the Oracle JHeadstart Team (idevcoe_nl@oracle.com), and send us the following information:

- The JHeadstart version you are using. You can see this in JDeveloper by going to the Help menu -> JHeadstart Documentation Index.
- The name of the form module that is causing the problem.
- The error message and error stack trace displayed in the dialog window (if any)
- Any log information written to the JDeveloper JFG log window
- The XML files created during processing for this module: _extracted.xml, _migratedAdfbc.xml and migratedJhs.xml file. Depending on the phase in which the error occurred, not all of these files might be available for the form module. Please send us the files that are available.
- Any other information that can help us understand your issue. For example, if you expected a different layout of the user interface, then attach screen shots of the original form, and the generated web page.
- **Note**: if you send zipped attachments (preferred), then make sure you rename extension .zip to something else (.zipp), otherwise your e-mail will be bounced by the Oracle Mail Server.

With this information, we might be able to reproduce the problem, and provide you with a patch that fixes the problem.

13.3.7. Processing the Same Form Multiple Times

You can run the JFG multiple times for the same form. When you do this the following happens:

- Existing entity objects will not be modified, only new attributes will be added if there are columns in the table that are not yet mapped to an attribute
- Existing view objects will not be modified, only new attributes will be added if there are items in the block that are not yet mapped to an attribute
- Existing application modules will not be modified, only new view object usages will be added if necessary
- The groups generated for the form in the JHeadstart Application Definition will not be changed at all. If you want the JFG to recreate the group definitions, you first need to change the top-level group of the form and all its detail groups.

13.4. Understanding the Outputs of the JHeadstart Forms2ADF Generator

This section will give you a short overview of the output and explains how this can be related to Oracle Forms elements. It also discusses possible changes you might need to make to the generated outputs.

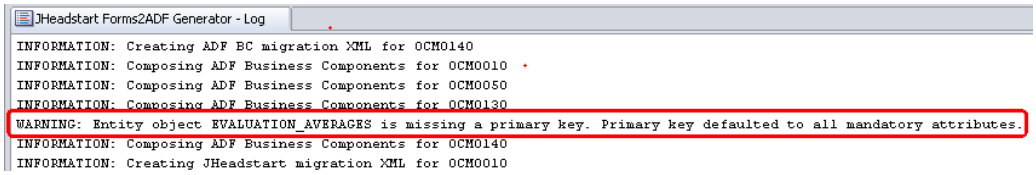
13.4.1. Generated ADF Business Components

13.4.1.1. Entity Objects

For each table or view used by an Oracle Forms data block, an Entity Object (EO) is created. These entity objects are shared across all forms modules: when multiple forms use the same table, only one entity object is created.

The information to create an EO is queried from the data dictionary tables. This means that attributes are created for each column in the table, regardless of whether the column is actually used in a forms block.

EO attributes are marked as key attributes based on the primary key definition in the data dictionary. If the table does not have a primary key, or the block is based on a view, then the JFG will default all mandatory attributes as primary key attributes. This is done because ADF Business Components requires each EO to have at least one key attribute. When this happens, a warning is written to the log window:



When you an encounter such a warning, you should check which attributes really make up the primary key, and change the Primary Key checkbox in the EO attribute editor accordingly.

13.4.1.2. Entity Association

Foreign Key constraints as found in the data dictionary are processed into **Entity Associations**. 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.

13.4.1.3. View Objects

View Objects (VO) are created for each block in the form that is based on a table or view. These "block" VO's are always based on the entity object that maps to the same table or view. For each data bound item within the block a VO attribute is created.

In older forms, lookup data (like the department name in an employee block) is typically displayed in block items not based on a column. The values in these unbound items are typically set through a POST-QUERY trigger that performs lookup queries to associated tables. Since the JFG does not parse any PL/SQL logic, these lookup values will not be visible in the ADF web application that can be generated using the outputs of the JFG.

However, when the block is updateable, it is quite likely that a List of Values (LOV) has been defined in the form that populates the foreign key item (for example department_id), as well as lookup items (for example department_name). If such an LOV is present, JFG will create so-called calculated attributes for the lookup items that are populated through an LOV. The calculated attribute gets a SQL expression that returns the lookup item value based on the Record Group Query associated with the LOV.

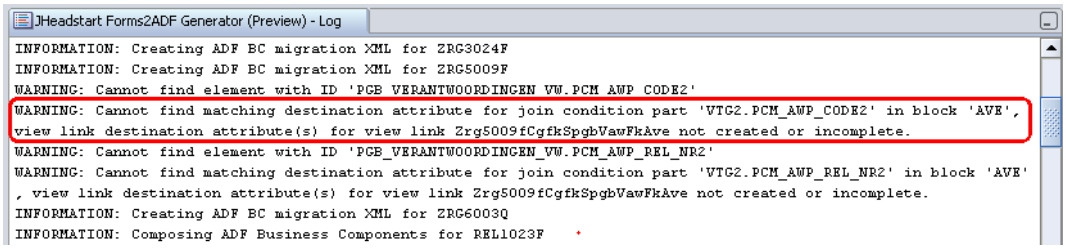
In addition for the VO's created for data blocks, read-only VO's, not based on an EO, are created for each Record Group Query found in the form. These read-only VO's are used as the data collection for the LOV groups created in the JHeadstart Application Definition.

The WHERE and ORDER BY clauses of the data blocks are also migrated to the View Object. Any references to block items are replaced with Query Bind variables, and in the JHeadstart Application Definition, the **Query Bind Arguments** group property is set up to populate these bind variables with the correct values. References to Forms globals and system variables cannot be replaced with bind variables, and therefore these references are changed to literal values by enclosing them in brackets, to keep the SQL query valid. When the JFG encounters such a reference to a Forms global or System variable, you are notified in the log window. Make sure you revisit these View Object and change the query, where clause or order by clause as needed.

13.4.1.4. View Links

When a relation has been defined between two data blocks in the form, the JFG attempts to create a view link between the View Objects created for the two blocks involved.

First, the JFG tries to find a foreign key constraint between the two tables, as defined in the data dictionary. If such a foreign key constraint exists, the view link is created based on this foreign key. If no foreign key is found, JFG parses the content of the forms **Relation Join Condition** property. If the value of this property exists of one or more conditions connected with the AND operator and each condition uses a simple '=' operator with two item references, JFG creates a view link based on the attributes created for these items. If the **Relation Join Condition** is more complex, or the JFG cannot find matching attributes for the items, the JFG will display a warning in the log window:



In such a situation, the view link will be created without source and destination attributes. You will need to add those attributes manually by double clicking on the view link.

13.4.1.5. Application Module

For each form, an application module is created that contains usages for all the View Objects created for the form. Each form-specific application module is added as a nested application module usage to the Root Application Module.

To make sure the ADF Business Components created are all valid, we recommend to run the ADF BC tester on the Root application module, available through the right-mouse-click menu on the application module. By running the tester, you can check:

- If your project compiles successfully. If not correct the compilation errors first.
- If all View Object queries have been brought forward in a correct manner, and return the correct data
- If you can make updates through the entity-based View Objects, which are all View Objects that originate from a Forms block.

13.4.2. Generated JHeadstart Application Definition File

The JFG also creates the JHeadstart Application Definition file. The sub sections below explain how and when each element type in the JHeadstart Application Definition file is created.

13.4.2.1. Groups

The JFG creates a **Group** for each block in the forms module. A top-level group is created for

- The block that is specified as the first navigation block in the form, or if not specified,
- The first block in the form that is not a detail block of another block

A detail-group is created for

- Each block that is a detail block of the block mapped to the parent group
- All other blocks that are not a detail block, they are created as detail group of the top-level group. When such a block is based on a table or view, the boolean **Dependent Data Collection** property is set to false, as there is no link with the data collection of the parent group.

For each LOV in the form that has an associated Record Group Query, a group is created with the boolean property **Use as LOV** set to true.

If the JFG created query bind parameters for the underlying View Object of a group, the **Query Bind Arguments** property of the group will be set accordingly. This ensures that the JHeadstart Application Generator will generate the appropriate configurations to populate these query bind parameters with the correct item values at runtime.

If a block is not based on a table or view, then the boolean group property **Bound to Model Data Collection** is set to false.

13.4.2.2. Items

For each item in a block, a corresponding item is added to the group that was created for the parent block. If the item is not based on a column, the item will have the **Databound** property set to false, otherwise it will be set to true, and the matching View Object attribute will be set in the **Attribute Name** property.

The item display type is set based on the item type of the source form item.

If the form item has an LOV associated the item display type will be set to “lov”, and a ListOfValues child element will be added to the JHeadstart item. The ListOfValues element will have child ReturnValue elements, one for each LOV Column Mapping in the form.

13.4.2.3. Region Containers, Item Regions and Group Regions

Based on the placement of an item:

- on a content canvas
- on a stacked canvas
- on a tab page
- within a framed graphic

and taking into account placement of parent and detail group items on the same canvas or tab page, or within the same framed graphic, the JFG will create the appropriate nested structure of region containers, item regions and group regions.

The number of layout columns set on an item region, is based on the first line of items within the region in the original form. For example, if 3 items have the same Y position coordinate within the tab page or graphic, then the **Columns** property will be set to 3, regardless of the number of items on subsequent lines.

13.4.2.4. Domains

A static domain will be created for the following form items:

- Checkbox Items: a domain with two allowable values, the checked and unchecked value will be created. If the unchecked value is not set, the value ‘N’ is taken as unchecked value, since ADF only supports boolean value bindings for checkboxes that require both a checked and unchecked value.
- List Items: for each list element an allowable value will be created within the domain
- Radio group items: for each radio button element an allowable value will be created within the domain

Note that the JFG will not create dynamic domains for drop down lists. This is not possible because Oracle Forms does not support Record Group Queries to be attached to drop down lists. To populate a drop down list with dynamic values in Oracle Forms, custom PL/SQL logic needs to be written. Since the JFG does not attempt to parse the PL/SQL logic, we cannot create dynamic domains for drop down lists. Subsequently drop down list items that are populated through PL/SQL will be created as a normal text item by the JFG. Of course, after you ran the JFG, you can easily change the text item into a drop down list and create the associated dynamic domain manually in the JHeadstart Application Definition Editor.

13.5. Handling Forms PL/SQL Logic

The JHeadstart Forms2ADF Generator allows you to "copy" the PL/SQL logic used in the form to the JHeadstart Application Definition, so you can easily see the logic in the editor, and determine what to do with it. Below we have listed common types of PL/SQL logic, with some suggestions on how you might handle it. Note that this is a high-level overview, not a detailed cookbook on how to handle each piece of PL/SQL logic. Always make sure you fully understand the PL/SQL logic before you take a final decision on how to re-implement it in the ADF/JHeadstart stack.

- Canvas and window management logic: this kind of logic can typically be ignored, as it is specific to how Oracle Forms works.
- Navigation logic: logic to navigate to detail windows within the same form can typically be ignored, since JHeadstart will generate buttons to navigate between parent and detail groups. Navigation logic to call other forms, passing along context parameter(s) used to query information in the called form can be implemented using the JHeadstart Deep Linking functionality. See chapter 6 "Generating User Interface Widgets" section 6.14 "Hyperlink to Navigate Context-Sensitive to Another Page (Deep Linking)" for more information.
- Logic to implement conditionally dependent items: with this we mean PL/SQL logic that changes the user interface properties like required, enabled, or visible for one or more "dependent" items based on the value(s) of one or more "depends on" items. In other words, this kind of logic creates dynamic user interfaces that change based on the values you enter. JHeadstart offers extensive declarative support for conditionally dependent items. See chapter 6 "Generating User Interface Widgets", section 6.12 "Conditionally Dependent Items" for more information.
- Business rule logic: logic that causes error or messages or dialogs to be displayed when the user enters invalid data, or logic that automatically updates other values either directly in the form or by executing SQL DML statements (change event rules). This kind of logic can be implemented in ADF Business Components, as described in the *Business Rules White Paper*, or can be moved to the database (in case it is not yet implemented there).

Reference: For information about enforcing business logic within the ADF BC Business Service, see the whitepaper Business Rules in ADF BC: otn.oracle.com/products/jdev/collateral/papers/10131/businessrulesinadfbctechnicalwp.pdf

13.5.1. Moving PL/SQL Logic to the Database

When analyzing the PL/SQL logic in older forms, you might encounter a lot of PL/SQL logic to implement business rules or computations that might well be executed inside the Oracle Database. Using the "Move to database" icon in the JHeadstart Application Definition Editor, which is enabled when you select a PL/SQL program unit or trigger in the navigator tree, you can easily move such logic to the database.

If you want to call out to PL/SQL procedures or functions that you moved to the database from a button in your ADF web page, you need to do the following:

- Create a custom business method in your application module class that calls the stored procedure or function, by following the instructions in the ADF Developer's Guide for Forms/4GL Developers, section 25.5 "Invoking Stored Procedures and Functions".

Generate the button to call the method by following the instructions in chapter 6 "User Interface Widgets", section 6.13 "Custom Button that Calls a Custom Business Method".

This page is intentionally left blank.

CHAPTER

14

JSF-ADF Page Lifecycle

This chapter provides an architectural overview of the Page Lifecycle in JSF-ADF applications and how JHeadstart plugs into this lifecycle.

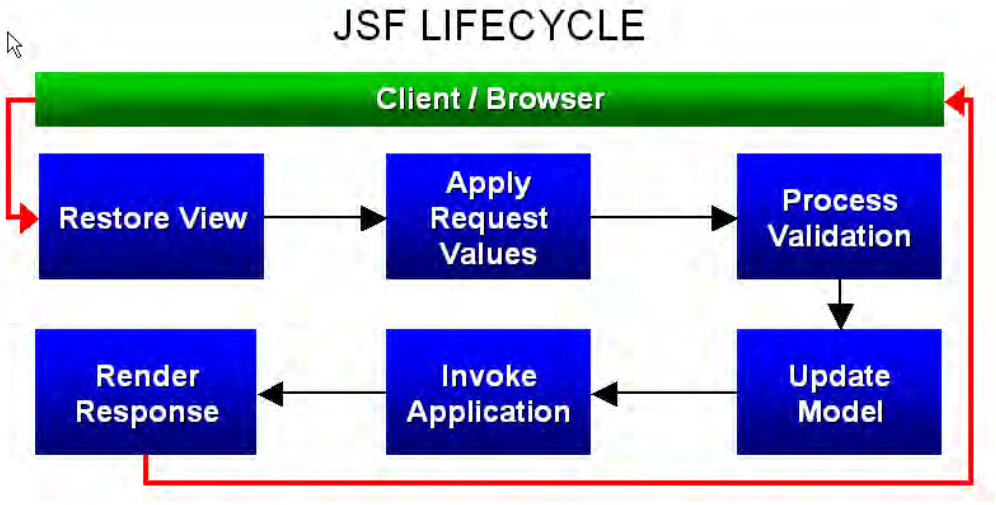
If you want to build more advanced applications using JSF, ADF and JHeadstart it is crucial that you understand what happens in which sequence when you submit a HTTP Request from a web page. The actions performed from the moment you submit a page, until the next page is displayed in the browser is called the *Request Processing Lifecycle* or *Page Lifecycle* or just *Lifecycle* in JSF terms.

The various types of actions performed when submitting a request are divided in so-called *Lifecycle Phases*. Standard JSF applications have 6 Lifecycle phases. When using ADF together with JSF, you will even have 9 Lifecycle phases. In the paragraphs of this chapter we will first explain the standard JSF Lifecycle phases, then the phases added by ADF, and finally we will discuss how JHeadstart extends the default behavior of some of the Lifecycle Phases.

Note that this section assumes you have a basic understanding of JSF concepts like UI Component, Action and Action Listener Property, Validator, Managed Bean and Navigation Rule. If you lack this background, we recommend that you first google for "Introduction to JSF" and follow one of the tutorials that are returned as search result.

14.1. JSF Lifecycle Phases

The standard JSF Page Lifecycle consists of 6 phases as shown in the picture below.



Each phase is briefly described below.

14.1.1. Restore View Phase

In this phase the tree of UI Component java classes is restored in memory, so that the next lifecycle phases can access the various UI Component class instances created for each UI component that is defined in the JSP page.

14.1.2. Apply Request Values Phase

In this phase, JSF populates the UI Component class with the value as submitted by the user. This always happens and is the reason that when you submit a JSF page with some input fields, by default the page will redisplay with the values as you entered them. So, finally, we do not have to do any coding at all to redisplay a page with the entered values! The UI Component class (re-)created for each input element in the Restore View phase will keep track of the submitted value.

14.1.3. Process Validation Phase

In this phase, validators and converters that might be defined in the JSP page will be executed for each UI Component. When validation or conversion fails, an error message is added to the stack of JSF messages. When validation and conversion is successful, the converted (also called "decoded") value is stored in the UI Component class. Whereas the submitted value is always a string (HTTP Request parameters are always strings) the decoded value can be of any type. For example, when a date converter is defined against an input element, the decoded value will be of type java.util.Date. In summary, the UI Component class of an editable input element can hold two values: the submitted value (accessible through `getSubmittedValue()`) and the decoded value (accessible through `getValue()`).

14.1.4. Update Model Phase

This phase is only applicable for editable UI Components that have the value property defined in the JSP page. The value property typically contains an EL expression that references a managed bean property, or in case of an ADF Model binding, the inputValue property of an ADF Value Binding class. In the Update Model phase, the setter method of the property specified in the EL expression is called with the decoded value of the UI Component. So, the following element in a JSP Page:

```
<h:inputText  … value="#{loginBean.username}"/>
```

causes the `setUsername()` method to be called in the java class defined in the managed bean definition named `loginBean`, stored in the faces-config.xml.
When using ADF bindings, you will typically see value properties like this:

```
<h:inputText …. value="#{bindings.LastName.inputValue}"/>
```

This causes the setInputValue method to be called on an ADF Model value binding class that is created at runtime by ADF. The next section explains in more detail how that works. For now, it is important to understand that JSF executes the Update Model phase in complete ignorance of ADF and the ADF Data Binding layer. JSF simply parses the EL expression in the value property of each UI component, and calls the corresponding setter method.

14.1.5. Invoke Application Phase

In this phase, JSF calls the methods specified in the *action* property or *actionListener* property of the JSF control used to submit the page (typically a button or hyperlink). When the action property is set, the outcome of the action property (either a hard coded literal value in the action property itself, or the return value of the action method referenced in the action property) is used to determine the next page that must be rendered. JSF looks up in the faces-config the navigation rules that match both the submitted page and this action outcome. Again, JSF executes the Invoke Application phase in complete ignorance of ADF and the ADF Data Binding layer. It simply calls the method specified in the EL expression of the action or actionListener property:

`<h:commandButton action="#{loginBean.authenticate}"/>` causes the `authenticate()` method to be called in the java class specified in the managed bean definition named `loginBean` in the faces-config.xml.

`<h:commandButton actionListener="#{bindings.Commit.execute}"/>` causes the `execute()` method to be called on the ADF action binding class created at runtime for the Commit binding.

14.1.6. Render Response Phase

In this phase, the UI Component renderers are invoked to create the HTML markup that is returned to the browser.

14.1.7. The Impact of the Immediate Property

The above sequence of lifecycle phases is the *default* JSF behavior. However, JSF is highly customizable. We can skip lifecycle phases programmatically or declarative, and we can add additional lifecycle phases as we will see in the next section. There is one simple property, the *immediate* property, that allows you to declaratively change the default

lifecycle sequence. When you set the immediate property to true on the command control that you use to submit the page, the Process Validation and Update Model Phases are skipped. A typical example where you use the immediate property is the Cancel button. When pressing Cancel, the user wants to abandon his current data entry task, any user entered values should be ignored, and he should not get any validation errors about required items, or other validation rules that might have been violated by already entered values. The immediate property is doing exactly what is needed in this case: it skips all validations and does not update the model.

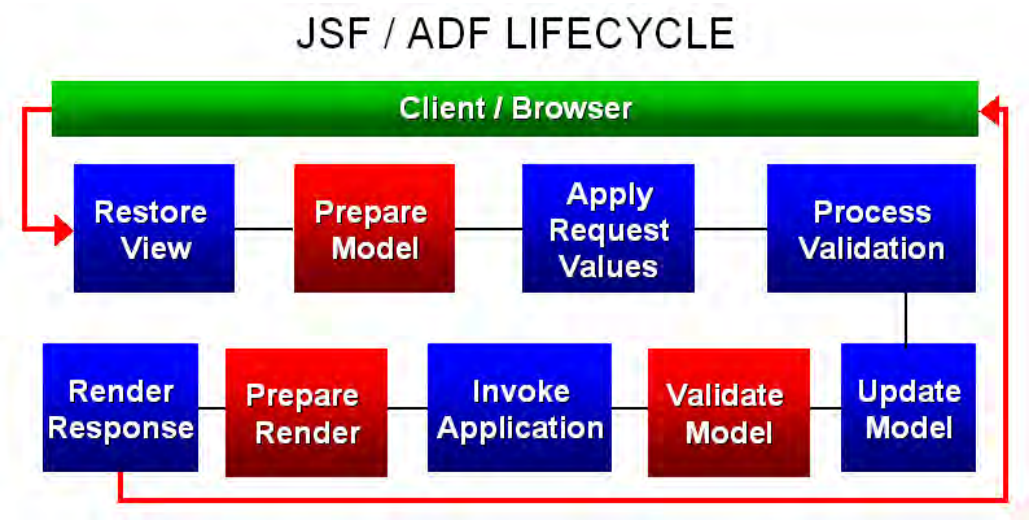
When using ADF Faces, you can use the immediate property to skip the client-side JavaScript validations, like requiredness of an item. This is quite useful, for example when using partial page rendering to implement functionality like conditionally dependent items. But remember, the Update Model phase is also skipped!

14.2. ADF-Specific Lifecycle Phases

As said before, JSF is highly customizable. You can plug in your own custom lifecycle phases if you like by using a so-called Phase Listener class. ADF uses this technique to integrate the ADF Model layer with JSF. If you create a simple drag and drop application, and then lookup the source of the faces-config.xml, you will see the following entry:

```
<lifecycle>
  <phase-listener>oracle.adf.controller.faces.lifecycle.ADFPhaseListener
  </phase-listener>
</lifecycle>
```

Through this ADFPhaseListener, ADF adds three more phases to the JSF Lifecycle, as shown in the picture below.



The **Prepare Model** and **Prepare Render** phases both handle the executables section in the page definition. The executables section contains two types of executables:

- Iterator executable that might cause the associated iterator (ViewObject Usage) to be (re-)queried.
- Invoke Action executable that might cause the associated action binding to be executed.

For example, you will have noticed that when you create a simple form page with ADF drag and drop, and you run the page, you see the data of the first row in the underlying table. This happens because when you dragged and dropped your data collection from the data control palette onto your page, an iterator binding was added to the executables section of your page definition. By default, an iterator executable queries the underlying ViewObject usage in the PrepareModel phase if it was not queried before. An executable has two properties that provide control over when the iterator is queried (or the action invoked), and in which phase: Prepare Model, Prepare Render or both. These properties are **Refresh** and **RefreshCondition**.

JHeadstart uses these properties to implement functionality like Deep Linking, Query Bind Params, and "No-Autoquery" Mode, as you will see later in this chapter.

Reference: For more information about using the Refresh and Refresh Condition properties, see the ADF Developer's Guide for Forms/4GL Developers, section 10.5.5:

http://download-west.oracle.com/docs/html/B25947_01/bcdcpal005.htm#BJECHBHF

The **Validate Model** phase validates all "dirty" ADF BC View Objects and Entity Objects. That is, those View Objects and underlying Entity Objects that have an attribute that is updated in the Model Update phase through a value binding that is associated with this attribute. In other words, this phase results in a call to the `validateEntity()` method of each Entity Object that has been changed during the Update Model phase. Typical model exceptions reported are required attributes without a value.

This phase should not be confused with the **Process Validations** phase, which is related to the View layer of your application. The Process Validations phase validates the submitted values of UI Components, based on validators defined against the UI component in the JSP page, and properties like "required". The Validate Model phase applies to your Business Service layer, it calls validation methods in the Business Service.

14.2.1. Customizing the ADF-JSF PageLifecycle

In the ADFPhaseListener class, you will find the following method:

```
protected PageLifecycle createPageLifecycle()
{
   return new FacesPageLifecycle();
}
```

As you can see, this method creates an instance of `oracle.adf.controller.faces.lifecycle.FacesPageLifecycle`. This class subclasses `oracle.adf.controller.v2.lifecycle.PageLifecycle`, which contains (amongst others) three methods, one method for each of the three custom phases added by ADF to the JSF Lifecycle:

- `prepareModel()`
- `prepareRender()`
- `validateModelUpdates()`

If you want to extend or override the default behavior of these ADF custom phases, for example to skip model validation, you can override one or more of these methods in your own subclass, and register your own Phase Listener class in the faces-config that instantiates your application. When using JHeadstart, this process is a bit easier, you do not need to create your own PhaseListener class as will be explained in the next section.

Attention: There are other methods that you can override as well, like `reportErrors()`. However, there are two methods that are useless to override: `processUpdateModel()`, and `processComponentEvents()`. These methods are never called in a JSF context., because they provide functionality that is part of the standard JSF Lifecyle:

- The JSF Update Model phase provides the same functionality as `processUpdateModel()`.
- The JSF Invoke Application phase provides the same functionality as `processComponentEvents()`.

The methods are there because the FacesPageLifecycle class extends a generic PageLifecycle class that is also used when building web applications without JSF. For example, when using Struts the ADF framework does call these methods.

14.3. JHeadstart Page Lifecycle

The JHeadstart Runtime includes its own lifecycle class `oracle.jheadstart.controller.jsf.lifecycle.JhsPageLifecycle` that subclasses `oracle.adf.controller.faces.lifecycle.FacesPageLifecycle`. As you will expect after reading the previous section, JHeadstart registers the JhsPageLifecycle class using its own Phase Listener class, `JhsADFPhaseListener`. When you generate your application with JHeadstart, the following entry is generated into a faces-config file named JhsCommon-beans.xml:

```
<lifecycle>
  <phase-listener>oracle.jheadstart.controller.jsf.lifecycle.JhsADFPhaseListener
  </phase-listener>
</lifecycle>
```

Attention: JHeadstart generates multiple faces-config files, to nicely organize the various elements that can be recorded in a faces-config file. You can set the names or directories of these files through service-level properties in the Application Definition, here is the list if you keep the default settings:

- The JhsCommon-beans.xml faces config contains the above <lifecycle> element, as well as generic bean definitions that are common to the whole web application. Even when you have multiple application definitions to generate your application, you still need only one JhsCommon-beans.xml.
- The "Main" faces-config.xml contains the navigation rules of your service (application definition). You can have multiple Application Definitions generated into the same faces-config.xml, or in different files when you change the Service-level property Main Faces Config.
- The "Group" faces-config files hold all the managed bean definitions used by the pages generated for a specific group.
- The Breadcrumb faces-config holds all the breadcrumb managed bean definitions for one service (application definition).

Of course, JSF needs to know it should load all these generated faces-config files. This is handled by a <context-param> element named javax.faces.CONFIG_FILES in the web.xml, which contains a comma-delimited list of all faces-config files. If this context parameter does not exist, JSF only loads the default faces-config, which should be named faces-config.xml, located in the WEB-INF folder.

If you go to the source of the JhsADFPhaseListener class, you will see the following method:

```
protected PageLifecycle createPageLifecycle()
{
  return (PageLifecycle)JsfUtils.getExpressionValue("#{jhsPageLifecycle}");
}
```

Rather than hardcoding the instantiation of a specific PageLifecycle class, as is done in the default ADFPhaseListener class (see previous section), JHeadstart uses a very powerful JSF technique: you can get hold of any java class defined as a managed bean, by using the same sort of EL expression as you use in your JSP pages to reference managed beans. The above expression will return an instance of the class that is defined

in one of the loaded faces-config files under the name “jhsPageLifecycle”. By default, JHeadstart generates the following managed bean definitions in the JhsCommon-beans.xml:

```
<managed-bean>
  <managed-bean-name>jhsPageLifecycle</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.lifecycle.JhsPageLifecycle
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>jhsWizardPageLifecycle</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.lifecycle.JhsPageLifecycle
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>validateADFModel</property-name>
    <value>false</value>
  </managed-property>
</managed-bean>
```

The second managed bean named “jhsWizardPageLifecycle” is not used by default but is generated for your convenience. This second bean also illustrates why using the technique of looking up the PageLifecycle class through a managed bean definition is so powerful: you can configure specific behaviors of the class without creating a subclass by specifying managed properties. For example, you can configure the “Transaction Completed” message or whether model validation should be performed.

By default, the generated jhsPageLifecycle managed bean references the `JhsPageLifecycle` class because this is the default value of the service-level property “Page Lifecycle class”. If you want to use a custom subclass, you can do so by creating a java class that extends `JhsPageLifecycle` and specify the name of this subclass as the value of the Page Lifecycle Class property.

If you want to use a specific PageLifecycle class for one specific group rather than the whole application, you can do this by setting the group-level Page Lifecycle class property. Unlike the service-level property by the same name, this property can contain an EL expression that references your custom lifecycle managed bean definition.

The `JhsPageLifecycle` class, together with the bean classes in the `oracle.jheadstart.controller.jsf.bean` package form the heart of the JHeadstart Runtime, together they implement the sophisticated runtime behaviors you can generate.