

ORACLE JHEADSTART 11g for ADF

RELEASE 11.1.2

DEVELOPER'S GUIDE

APRIL 2012

ORACLE®

JHeadstart Developer's Guide

Copyright © 2012, Oracle Corporation

All rights reserved.

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

Contributors: Pieter Biemond, Sigrid Gylseth, Bouke Nijhuis, Evert-Jan de Bruin, Paco van der Linden

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

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

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

CONTENTS

CHAPTER 1	GETTING STARTED	1-1
	1.1. Introduction into JDeveloper, ADF and JHeadstart	1-2
	1.1.1. Oracle JDeveloper	1-2
	1.1.2. Oracle Application Development Framework (ADF)	1-2
	1.1.3. What is Oracle JHeadstart?	1-3
	1.2. Roadmap to Developing ADF Applications using JHeadstart	1-5
	1.2.1. Set Up Project for Team-Based Development	1-5
	1.2.2. Create Business Service using ADF Business Components	1-6
	1.2.3. Design and Generate Web Pages	1-7
	1.2.4. Design and Generate Security Structure	1-8
	1.2.5. Customize Generated Web Tier	1-8
CHAPTER 2	SET UP PROJECT FOR TEAM-BASED DEVELOPMENT	2-1
	2.1. Setting Up Version Control System	2-2
	2.1.1. Version Control Models	2-2
	2.1.2. Requirements for a Good Version Control System	2-3
	2.1.3. Which Files to Version?	2-4
	2.2. Setting up Structure of JDeveloper Workspace and Projects	2-6
	2.2.1. Installing JDeveloper	2-6
	2.2.2. Identify Subsystems within your Application	2-6
	2.2.3. Creating a Workspace and Projects	2-6
	2.2.4. Creating Database Connection	2-9
	2.2.5. Initializing Model Project for Business Components	2-10
	2.2.6. Optimizing ADF BC for Team Development	2-10
	2.2.7. Switching off Default Creation of ADF BC Java classes	2-10
	2.2.8. Enabling ViewController Project for JHeadstart	2-11
	2.3. Organizing JHeadstart Service Definition Files	2-12
	2.3.1. Naming Conventions for File Location Properties	2-12
	2.4. Packaging JHeadstart-Generated ViewController Project as ADF Library	2-13
	2.4.1. Remove JhsCommon-beans.xml from the ADF Library jar file	2-13
	2.4.2. Add reference to resource bundle of ADF Library project	2-13
	2.4.3. Import JHeadstart service definitions in main ViewController project	2-14
	2.5. Defining Java Package Structure and Other Naming Conventions	2-16
	2.5.1. Java Packages	2-16
	2.5.2. Naming ADF Business Components	2-17

CHAPTER 3	CREATING ADF BUSINESS COMPONENTS	3-1
3.1. Setting Up ADF BC Base Classes.....		3-2
3.1.1. Using CDM RuleFrame		3-4
3.2. Creating the Entity Object Layer.....		3-5
3.2.1. Review Database Design		3-5
3.2.2. Creating First-Cut Entity Objects and Associations.....		3-5
3.2.3. Renaming Entity Objects and Associations		3-6
3.2.4. Generating Primary Key Values		3-7
3.2.5. Setting Entity Object Attribute Properties used by JHeadstart		3-8
3.2.6. Implementing Business Rules		3-9
3.3. Creating View Objects and Application Modules.....		3-11
3.3.1. Creating View Objects and View Links.....		3-11
3.3.2. Renaming View Objects and View Links		3-11
3.3.3. Inspecting and Setting Key Attributes of a View Object		3-11
3.3.4. Setting View Object Control Hints		3-12
3.3.5. Determining the Order of Displayed Rows.....		3-12
3.3.6. Creating Calculated or Transient Attributes.....		3-13
3.3.7. Setting Up Master-Detail Synchronization		3-13
3.3.8. Defining View Links and View Object Usages for Lookups		3-15
3.3.9. Testing the Model.....		3-15
CHAPTER 4	USING JHEADSTART	4-1
4.1. Understanding the JHeadstart Application Generator Architecture		4-2
4.1.1. Input Output.....		4-3
4.2. Using the JHeadstart Enable Project Wizard		4-4
4.2.1. Enabling JHeadstart on a new project.....		4-4
4.2.2. Enabling JHeadstart on an existing project.....		4-5
4.2.3. Re-enabling JHeadstart on a project		4-6
4.3. Using the Create New Service Definition Wizard.....		4-8
4.3.1. Step 1: Choose Data Control.....		4-9
4.3.2. Step 2: Choose Basic Settings.....		4-10
4.3.3. Step 3: Choose List of Values Options		4-11
4.3.4. Step 4: Choosing Search Style		4-16
4.4. Using the Application Definition Editor		4-18
4.4.1. Maintaining the Application Definition		4-18
4.4.2. Application		4-21
4.4.3. Service		4-21
4.4.4. Groups		4-22
4.4.5. Items		4-22
4.4.6. Lists of Values		4-22
4.4.7. Regions		4-23
4.4.8. Detail Groups.....		4-24
4.4.9. Domains		4-25
4.4.10. Manipulating Objects.....		4-25
4.4.11. Novice Mode and Expert Mode.....		4-28
4.4.12. Synchronize Group with Underlying View Object		4-28
4.5. Running the JHeadstart Application Generator.....		4-30
4.6. Running the Generated Application.....		4-32

4.6.1. TroubleShooting	4-32
4.6.2. Dealing with Code Segment Too Large Error	4-33
4.7. What was Generated for What Purpose	4-35

CHAPTER 5 GENERATING PAGE LAYOUTS 5-1

5.1. Choosing a Group Usage	5-2
5.1.1. Stand-alone Pages	5-2
5.1.2. Region with Page Fragments (Recommended)	5-2
5.1.3. Reusing Groups	5-5
5.1.4. List of Values Window	5-13
5.1.5. Showing a Group in a Popup Window.....	5-13
5.2. Creating Form Pages.....	5-14
5.2.1. Displaying an Item at the Right of Another Item	5-15
5.2.2. Hiding Items on the Form Page	5-16
5.2.3. Create and Update Mode in Form Layout	5-16
5.2.4. Controlling the Page Title in Form Layout	5-18
5.3. Creating Select-Form Pages.....	5-19
5.4. Creating Table Pages	5-20
5.4.1. Using Table Layout and Stretching	5-20
5.4.2. Using a Table Overflow Area.....	5-22
5.4.3. Hiding Items in a Table	5-24
5.4.4. Allowing the User to Sort Data in a Table Page	5-24
5.4.5. Adding Summary Information to a Table	5-24
5.4.6. Change Table-Related ADF Business Components Settings	5-25
5.5. Creating Table-Form Pages	5-27
5.6. Creating Master-Detail Pages	5-28
5.6.1. Master-Detail on Separate Page.....	5-28
5.6.2. Master-Details on Same Page	5-29
5.7. Creating Tree Layouts.....	5-34
5.7.1. Choosing a Tree Layout Style.....	5-34
5.7.2. Generating a Basic Tree.....	5-35
5.7.3. Variation: Basic Tree with non-clickable nodes	5-41
5.7.4. Variation: Recursive Tree.....	5-42
5.7.5. Variation: Recursive Tree with Limited Set of Root Nodes	5-43
5.7.6. Variation: Tree showing only Children of selected Parent	5-45
5.8. Creating Shuttle Layouts.....	5-48
5.8.1. Creating Parent Shuttles.....	5-48
5.8.2. Creating Intersection Shuttles	5-50
5.8.3. Understanding How JHeadstart Runtime Implements Shuttles.....	5-51
5.9. Creating Wizard Layouts	5-54
5.9.1. Launching a Wizard using New Button	5-55
5.10. Controlling Page Layout Using Region Containers, Item and Group Regions	5-57
5.10.1. Using Item Regions	5-58
5.10.2. Using Group Regions.....	5-60
5.10.3. Mixing Item Regions and Group Regions.....	5-61
5.10.4. Generating Content in a Popup Window	5-63

GENERATING USER INTERFACE WIDGETS	6-1
6.1. Specifying the Prompt.....	6-2
6.2. Default Display Value.....	6-3
6.2.1. Using EL expressions	6-3
6.3. Display Type.....	6-4
6.4. Generating a Text Item	6-8
6.4.1. Define Item Display Width and Height.....	6-8
6.4.2. Setting Maximum Length.....	6-9
6.5. Generating a Dropdown List.....	6-10
6.5.1. Static Dropdown List based on a Static Domain.....	6-10
6.5.2. Translation of Static Domains.....	6-10
6.5.3. Dynamic Dropdown List based on a Dynamic Domain	6-11
6.6. Generating a Radio Group	6-13
6.6.1. Static Radio Group based on a Domain	6-13
6.6.2. Translation of Static Domains.....	6-13
6.6.3. Dynamic Radio Group based on a Dynamic Domain	6-13
6.7. Generating a List of Values (LOV).....	6-15
6.7.1. ADF Model LOV vs. Custom JHeadstart LOV	6-15
6.7.2. Creating an ADF Model LOV	6-17
6.7.3. Creating a custom JHeadstart LOV.....	6-17
6.7.4. Creating a (reusable) LOV group.....	6-18
6.7.5. Linking a Reusable LOV group to an item	6-18
6.7.6. Defining an LOV on a display item	6-19
6.7.7. Use LOV for Validation.....	6-23
6.7.8. Selecting multiple values in a List of Values	6-24
6.7.9. Passing Parameters to an LOV.....	6-25
6.7.10. Understanding How JHeadstart Runtime Implements List Of Values	6-26
6.8. Generating a Date (time) Field.....	6-29
6.8.1. Specifying Display Format for Date and Datetime Field	6-29
6.9. Generating a Checkbox.....	6-30
6.9.1. Generating a checkbox based on a Boolean attribute	6-31
6.10. Generating a Button Item.....	6-32
6.10.1. Positioning of Buttons.....	6-32
6.10.2. Appearance of Buttons.....	6-32
6.10.3. Executing a Button Action.....	6-33
6.10.4. Calling a PL/SQL Procedure or Function From a Button	6-37
6.11. File Upload, File Download, Showing Image Files, and Playing Audio Files	6-39
6.11.1. Combining File Display Options.....	6-40
6.11.2. Showing Properties of Uploaded Files.....	6-41
6.12. Generating a Graph.....	6-43
6.13. Conditionally Dependent Items.....	6-44
6.13.1. Using the Depends On Item(s) Property	6-44
6.13.2. Cascading Lists	6-46
6.13.3. Row Specific Dropdown Lists in Table	6-47

9.1.4. Customizing the Static Menu Layout	9-4
9.2. Dynamic Menu Structure	9-6
9.2.1. Creating the Database Tables.....	9-6
9.2.2. Enabling Dynamic Menus.....	9-7
9.2.3. Defining the Menu Structure At Runtime	9-9
9.2.4. Using a Dynamic Menu Tree Layout	9-10
9.2.5. Linking a User Interface Skin to a Module	9-10
9.3. Using Dynamic Tabs when Opening a Menu Item.....	9-12
9.3.1. Enabling Dynamic Tabs.....	9-12
9.3.2. Marking the Current Tab Dirty	9-13
9.3.3. Displaying Initial Tabs at Startup	9-13
9.3.4. Launching a New Tab from the Current Tab Task Flow	9-14
9.3.5. Closing a Dynamic Tab.....	9-14

CHAPTER 10

APPLICATION SECURITY 10-1

10.1. Understanding and Choosing Security Options with ADF and JHeadstart	10-2
10.1.1. Authentication and Authorization Type	10-2
10.1.2. Secure All Pages	10-3
10.1.3. Authorize Using Group Permissions	10-4
10.2. JHeadstart Security Tables and Security Administration Screens.....	10-6
10.2.1. Creating the Database Tables.....	10-6
10.2.2. Generating Security Administration Pages	10-7
10.3. Using ADF/JAAS for Authentication.....	10-9
10.3.1. Changes in web.xml.....	10-9
10.3.2. Login Page and Login Bean.....	10-10
10.3.3. Logout Button	10-10
10.3.4. Default Users and Roles in jazn-data.xml	10-10
10.4. Using Custom Authentication	10-11
10.4.1. JHeadstart Authentication Filter	10-11
10.4.2. Nested JhsModelService Application Module.....	10-11
10.4.3. Login Page and Login Bean.....	10-11
10.4.4. Logout Button	10-12
10.5. Restricting Access to Groups based on Authorization Information	10-13
10.5.1. Restricting Group Access using Permissions	10-13
10.5.2. Nested JhsModelService Application Module.....	10-14
10.5.3. JHeadstart Authorization Proxy	10-14
10.6. Restricting Group And Item Operations based on Authorization Information.....	10-17
10.6.1. Restricting Group Operations using Permissions.....	10-17
10.6.2. Restricting Item Operations	10-18
10.7. Using Your Own Security Tables	10-19
10.7.1. Changes when Using Custom Authentication	10-19
10.7.2. Changes when Using Custom Authorization and/or Permissions	10-19
10.7.3. Changes to SQL Script Templates	10-19

CHAPTER 11

INTERNATIONALIZATION AND USER ASSISTANCE 11-1

11.1. National Language Support in JHeadstart.....	11-2
11.1.1. Which Locale is Used at Runtime.....	11-3

11.1.2. Supported Locales	11-3
11.1.3. Adding a non-supported Locale.....	11-4
11.2. Using a Database Table to Store Translatable Strings	11-5
11.2.1. Creating the Database Tables	11-5
11.2.2. Running the JHeadstart Application Generator	11-6
11.2.3. Running the Application	11-7
11.3. Runtime Implementation of National Language Support.....	11-9
11.4. Error Reporting.....	11-11
11.5. Outstanding Changes Warning.....	11-12
11.6. Using Online Help	11-13
11.6.1. Using the Resource Bundle Online Help Provider	11-13
11.6.2. Using the Help Text and Instruction Text Property	11-14
11.6.3. Using Other Online Help Providers.....	11-15
11.7. Using Function Keys.....	11-17
11.7.1. What Gets Generated When Enabling Function Keys	11-18
11.7.2. What Happens When You Press a Function Key.....	11-18
11.7.3. Customizing the Function Key Implementation.....	11-19
11.7.4. Adding a New Function Key	11-20
11.7.5. Understanding the Difference Between Function Keys and Access Keys	11-21

CHAPTER 12

CUSTOMIZING GENERATOR OUTPUT	12-1
12.1. Recommended Approach for Customizing JHeadstart Generator Output.....	12-2
12.2. Understanding Generator Architecture and Generator Templates	12-4
12.2.1. Velocity and the Velocity Template Language.....	12-4
12.2.2. Understanding the JHeadstart Metadata Model	12-4
12.2.3. Referencing JHeadstart Metadata Model in Velocity Templates.....	12-6
12.2.4. Understanding Application Generators.....	12-7
12.2.5. Configuration of Generator Classes	12-9
12.2.6. Structure of Generator Templates.....	12-10
12.3. Creating Custom Templates	12-13
12.3.1. Creating a Custom Template File	12-13
12.3.2. Finding Out Which Generator Templates Are Used	12-15
12.3.3. Grouping Custom Templates	12-15
12.3.4. Writing Reusable Custom Templates.....	12-16
12.3.5. Customizing JHeadstart Macro's.....	12-20
12.3.6. Coding and Debugging Tips	12-21
12.4. Customizing Pages.....	12-23
12.4.1. Customizing Page Templates.....	12-23
12.4.2. ADF Faces Skinning	12-25
12.4.3. Changing the Page Template at Runtime	12-27
12.4.4. Adding Custom Properties to a Generated Item	12-29
12.4.5. Adding Custom Item Display Type	12-30
12.5. Customizing Task Flows.....	12-33
12.5.1. Understanding Generated Task Flow Structure	12-33
12.5.2. Customizing the Task Flow Template	12-35
12.5.3. Customizing the Task Flow Template Reference	12-36

12.5.4. Adding Custom Task Flow Parameters	12-36
12.5.5. Adding Custom Managed Beans	12-36
12.5.6. Customizing the Main Router Activity	12-38
12.5.7. Adding Custom Task Flow Activities	12-38
12.5.8. Adding Custom Control Flow Rules and Cases	12-38
12.5.9. Preventing Task Flow Generation	12-39
12.6. Customizing Output of the File Generator	12-40
12.6.1. How to customize the content of a file generated by the file generator	12-40
12.6.2. How to stop a file from being generated by the file generator	12-40
12.6.3. How to generate additional custom files	12-41
12.7. Customizing Page Definitions	12-42
12.7.1. Controlling Generation of Value Bindings	12-42
12.7.2. Preserving Bindings Added using a Drag-and-drop Action	12-43

CHAPTER 13 RUNTIME PAGE CUSTOMIZATIONS 13-1

13.1. Creating the Database Tables	13-2
13.2. Enabling Runtime Usage of Flex Items	13-4
13.2.1. Creating a Flexible Region	13-4
13.2.2. Running the JHeadstart Application Generator	13-5
13.3. Defining Flex Items At Runtime	13-7
13.4. Creating an Item with Display Type Flex Region	13-10
13.5. Internationalization and Flex Items	13-11

CHAPTER 14 FORMS2ADF GENERATOR 14-1

14.1. Introduction into JHeadstart Forms2ADF Generator (JFG)	14-2
14.1.1. Added Value of JHeadstart Forms2ADF Generator	14-3
14.2. Roadmap	14-5
14.3. Running the JHeadstart Forms2ADF Generator (JFG)	14-7
14.3.1. Select Forms Modules	14-7
14.3.2. Select Form Elements to be Excluded from Processing	14-8
14.3.3. Select Database Connection	14-9
14.3.4. ADF Business Components Settings	14-11
14.3.5. JHeadstart Settings	14-13
14.3.6. Processing the Selected Forms	14-14
14.3.7. Troubleshooting	14-15
14.3.8. Processing the Same Form Multiple Times	14-17
14.4. Understanding the Outputs of the JHeadstart Forms2ADF Generator	14-18
14.4.1. Generated ADF Business Components	14-18
14.4.2. Generated JHeadstart Service Definition File	14-21
14.5. Common Steps After Running the Forms2ADF Generator	14-24
14.5.1. Checking and Fixing the Model project	14-24
14.5.2. Checking and Enhancing the JHeadstart Service Definition	14-24
14.5.3. Handling Forms PL/SQL Logic	14-25

CHAPTER 15	ORAFORMSFACES GENERATOR	15-1
15.1. Introduction into OraFormsFaces™		15-2
15.1.1. Added Value of the JHeadstart OraFormsFaces Generator		15-2
15.2. Running the JHeadstart OraFormsFaces Generator (OFFG)		15-3
15.2.1. Select Forms Modules		15-3
15.2.2. Generator Settings		15-5
15.2.3. Processing the Selected Forms.....		15-5
15.2.4. Troubleshooting.....		15-7
15.3. Understanding the Outputs of the JHeadstart OraFormsFaces Generator.....		15-8
15.3.1. Create Application Module If Needed.....		15-8

Getting Started

Developing complex transactional applications on the Java Enterprise Edition (JEE) platform is not a straightforward task. Java and 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 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 for ADF.

<http://download.oracle.com/consulting/jhstutorial11112.pdf>

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 ADF 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 applications 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.

Application Development Framework (ADF) Fusion Stack

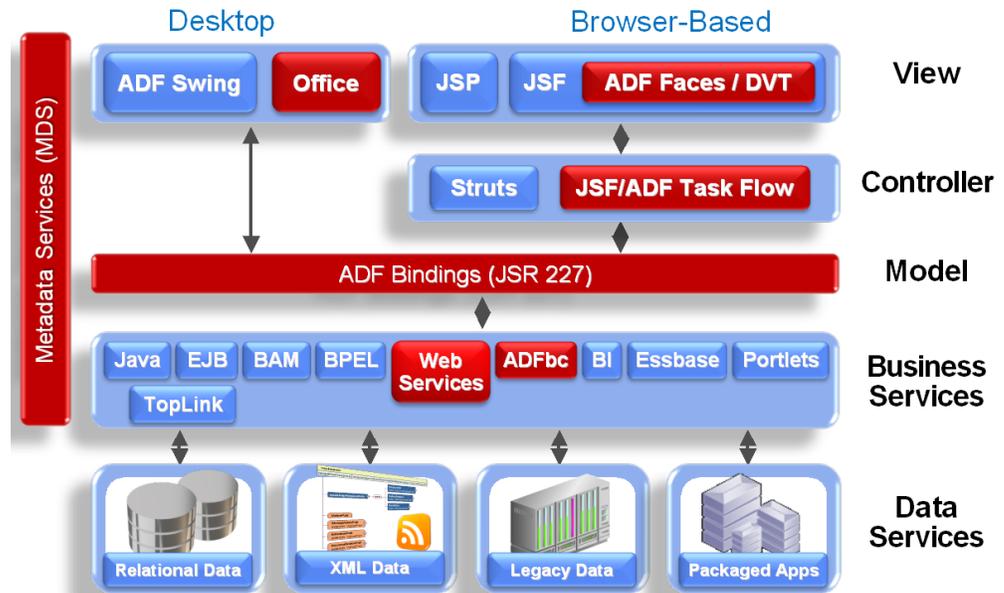


Figure 1-1 Oracle ADF Architecture



Oracle ADF on OTN. Overview, Online Demo's, Tutorials, Developer Guides, White Papers, How-to's and more:

<http://www.oracle.com/technology/products/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/ADFbc config files), View (ADF Faces pages/fragments), 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 Components, as well as the XML meta-data (Service 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 developer's 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: Fusion Developer’s Guide for Oracle Application Development Framework section 12.3. *Creating a Layer of Framework Extensions.*:

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/bcadvgen.htm#BABFDHBB

1.2.2.2. Create Entity Objects and Associations



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



Web Reference: Fusion Developer’s Guide for Oracle Application Development Framework chapter 4: Creating a Business Domain Layer Using Entity Objects.

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/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: Fusion Developer’s Guide for Oracle Application Development Framework, chapter 5: Defining SQL Queries using View Objects

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/bcquerying.htm#sm0070



Web Reference: Fusion Developer’s Guide for Oracle Application Development Framework Release, chapter 6: Testing View Instance Queries

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/bcqueryresults.htm#BEIHCB CF

1.2.2.4. Create Application Modules



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



Web Reference: Fusion Developer’s Guide for Oracle Application Development Framework, chapter 9: Implementing Business Services with Application Modules

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/bcservices.htm#sm0203

1.2.2.5. Implement Business Rules



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



Web Reference: Fusion Developer’s Guide for Oracle Application Development Framework, chapter 7: Defining Validation and Business Rules Declaratively
http://docs.oracle.com/cd/E24382_01/web.1112/e16182/bvalidation.htm#sm0231



Web Reference: Fusion Developer’s Guide for Oracle Application Development Framework, chapter 7: Defining Validation and Business Rules Programmatically
http://docs.oracle.com/cd/E24382_01/web.1112/e16182/bcrules.htm#CIHBBDDDB

1.2.3. Design and Generate Web Pages

1.2.3.1. Understand JHeadstart Generator Architecture 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 Service Definition Wizard”, “Using the Application Definition Editor”.



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

1.2.3.3. Configure Internationalization Options



Reference: Chapter 11 “Internationalization and User Assistance”, 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: Fusion Developer’s Guide for Oracle Application Development Framework, chapter 35: Enabling ADF Security in a Fusion Web Application.

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/adding_security.htm#BGBGJEAH

1.2.4.2. Implement User Authentication



Reference: Chapter 10 “Application Security”, sections “Using ADF/JAAS 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”

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 12 “Customizing Generation Output”, section “Recommended Approach for Customizing JHeadstart Generator Output”

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



Web Reference: Web User Interface Developer's Guide for Oracle Application Development Framework.

http://docs.oracle.com/cd/E24382_01/web.1112/e16181/toc.htm



Web Reference: Fusion Developer's Guide for Oracle Application Development Framework.

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm

1.2.5.3. Move Post-Generation Changes to Custom Templates



Reference: Chapter 12 "Customizing Generator Output", section "Creating Custom Templates"

1.2.5.4. Customizing Page Look and Feel



Reference: Chapter 12 "Customizing Generator Output" section "Customizing Pages"



Web Reference: Web User Interface Developer's Guide for Oracle Application Development Framework, chapter 28 "Customizing the Appearance Using Styles and Skins".

http://docs.oracle.com/cd/E24382_01/web.1112/e16181/af_skin.htm#BAJFEFCJ

1.2.5.5. Customizing Task Flows



Reference: Chapter 12 "Customizing Generator Output" section "Customizing Task Flows"

1.2.5.6. Customizing Page Definitions



Reference: Chapter 12 "Customizing Generator Output" section "Customizing Page Definitions"

1.2.5.7. 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 service 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 Good Version 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 back when a version conflict is detected which cannot be solved by an automatic merge. In other words, either all files are committed successfully, 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, committing 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. Managing ADF development across multiple applications with SubVersion

<http://www.oracle.com/technetwork/developer-tools/jdev/svn-adfsharedlibs-084371.html>



Using JDeveloper with SubVersion. Team-development using SubVersion – best practices

<http://www.oracle.com/technetwork/developer-tools/jdev/teamdevsubversion-089001.html>

The JHeadstart team has successfully 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 adfc-config diagram files, with extension “.adfc_diagram”. When generating your application with JHeadstart, the adfc-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 exclude 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.

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 WebLogic server.

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

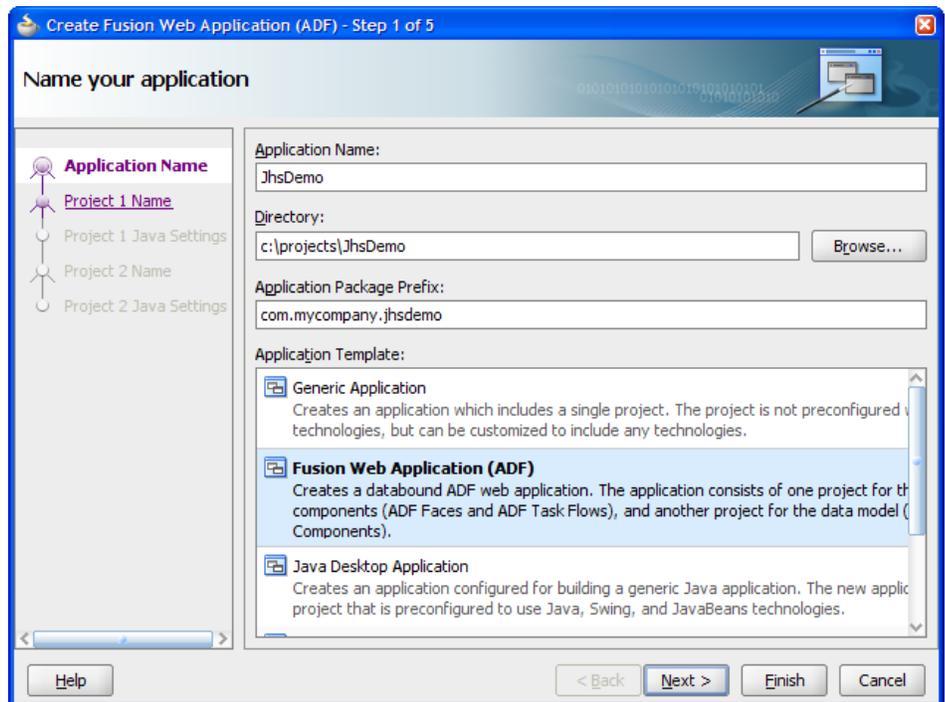
It is good practice to organize your application into logical subsystems. These subsystems can be used as a basis for

- 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.

- Select the Application Navigator (choose menu option View – Application Navigator)
- Choose 'New Application ...' from the dropdown list at the top of the Application Navigator.
- 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 **Fusion Web Application [ADF]** from the choice 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 creating 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.



Fusion Developer’s Guide for ADF, chapter 38 “Reusing Application Components”. Includes instructions on importing business components into another project.

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/reusing_components.htm#BABCHHHJ

- The application is very large and can be divided into subsystems with few dependencies on each other. Separate teams of developers work on each subsystem. 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 used as described before.

For a large application, you basically have two options for setting up the ViewController layer:

- Use multiple ViewController projects in combination with ADF Libraries
- Create one ViewController project and use working sets to define “views” over your project for each subsystem.

2.2.3.1. Using Multiple ViewController Projects

You can have multiple “helper” ViewController projects, and one “main” ViewController project. The “main” ViewController project is used to create the WAR deployment file. The “helper” ViewController projects are used to develop the various subsystems of the application, packaged as bounded task flows, which are then imported into the “Main” ViewController project using ADF Libraries. See the above reference to chapter 33 of the *Fusion Developer’s Guide for ADF* for more general information. See section [Packaging JHeadstart-Generated ViewController Project as ADF Library](#) in this chapter for JHeadstart-specific steps to make when using ADF Libraries.



ADF – Enterprise Methodology Group. The ADF – EMG is a group of ADF users, both experts and novice users, both Oracle employees as well as customers and partners that discuss anything beyond the typical “How do I” questions that are more suited for the JDeveloper discussion forum. There are some interesting ADF-EMG discussions about organizing large ADF projects.

<https://groups.google.com/forum/?fromgroups#!forum/adf-methodology>

2.2.3.2. Using Working Sets

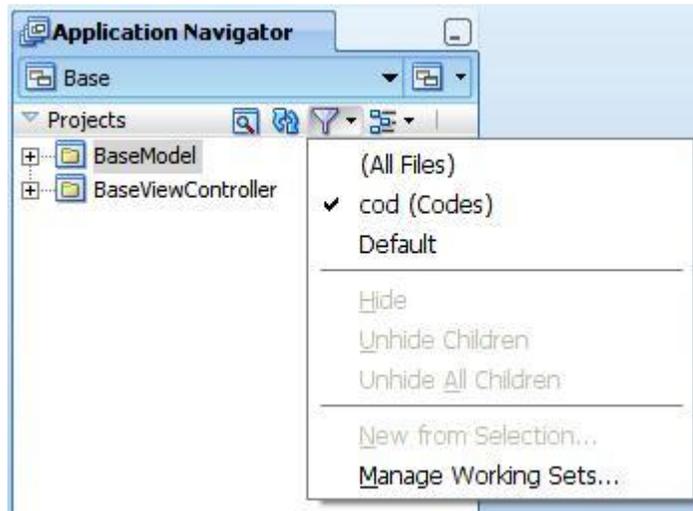


JDeveloper Online Help “Managing Working Sets”. Includes instructions on creating working sets.

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:

- Navigate
- Make
- Build/rebuild
- Search
- Find usages

- Search in Files



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 [JDevWLSHome]/jdeveloper/system/systemXXX/o.ide/projects folder and find the .jws file starting with the name of your JDeveloper Application, for example Base08345889adfc01e0ffd7dd7bfa1c6234.jws for an application called Base.
- Open this file, and you will see XML code starting with <hash n="working-sets"> that specifies your working sets.

Now you can either copy this piece of xml code, e-mail it to your colleagues and they can paste it in their application working sets file, or you can make sure that you all use the same application working sets file that is available from a common source. The latter is explained below.

- Copy (and possibly rename) this application working sets file to a location where it can be maintained centrally, but everyone has a local copy (for example in your Subversion repository).
- Tell everyone to change the relevant entry in their own [JDevWLSHome]/jdeveloper/system/systemXXX/o.ide/projects/index2.xml to let it point to the new file. (And don't forget to do that yourself as well ;-)

2.2.4. Creating Database Connection

You will need to create a Database Connection to the schema that contains the database tables of your application. In JDeveloper 11 you can create a connection per application file, which is recommended over IDE (user) specific connections.

Click on View - Database Navigator to show the connections you have. Right click your application file, and select 'New connection'. The connection you create there, will appear in the file [ApplicationRoot]/.adf/META-INF/connections.xml and can thus be stored in the application itself (rather than a user specific JDeveloper directory).

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 | Options panel, make sure the property named 'Copy Package XML Files to Class Path' is unchecked.

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.



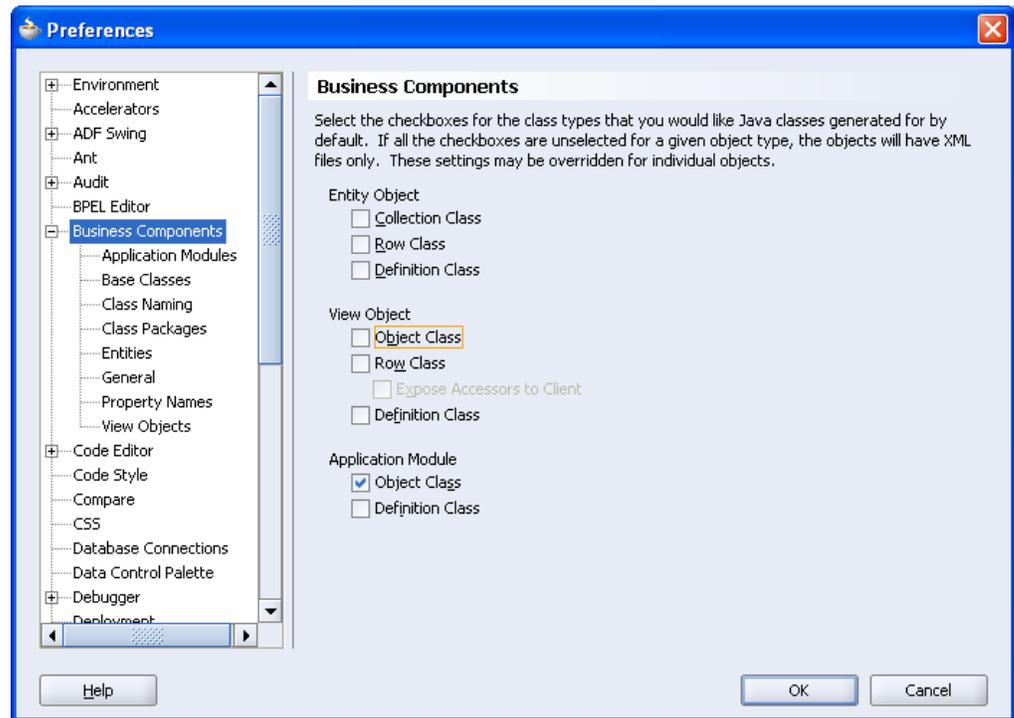
Suggestion: For existing projects, you can also edit this property at any time by unchecking it on the Business Components | Options panel of the project properties.

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 object 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 Service Definition Files

As explained in chapter 4 “Using JHeadstart Add-Ins”, the JHeadstart Application Generator is driven by the Application and Service Definition files that holds the generator metadata. Since release 11 of JHeadstart it is now possible to separate application-wide settings (stored in a file called JHeadstartApplicationDefinion.xml) from service specific settings (stored in files like MyServiceDefinition.xml).

Because of this separation, there is less need to keep settings ‘synchronized’ among Service Definition files, making team development and file merging an easier task.

We recommend creating multiple service definition files, even when working alone. The only exception would be a really small application. A typical approach is to create one service definition file for each logical subsystem. Since a service definition file is based on one data control (Application Module), the structure of your service definition files will typically follow the structure of your ADF BC application modules, which in turn should map your subsystem structure.

2.3.1. 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 that can be set at the service-level of a service definition file.

Here are suggested naming conventions.

Property	Value
Service Adfc Config	<i>/WEB-INF/adfc-config-<code><subsystem></code>.xml</i> <i>For example:</i> <i>/WEB-INF/adfc-config-hr.xml</i>
Group Adfc Config Directory	<i>/<code><subsystem></code>/adfcConfig/</i>
UI Pages Directory	<i>/<code><subsystem></code>/pages/</i>
Page Includes Directory	<i>/<code><subsystem></code>/regions/</i>

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

 **Attention:** Adfc Config XML files that are located outside the WEB-INF directory can be viewed in the browser. If you want to prevent this for security reasons, you should make subsystem subdirectories under the WEB-INF directory for the Adfc Config files.

2.4. Packaging JHeadstart-Generated ViewController Project as ADF Library

If you decide to use multiple ViewController projects as described before, and create ADF Libraries for these projects and add these ADF libraries to the main ViewController project, then there are some JHeadstart specific changes you need to make:

- Remove JhsCommon-beans.xml from the ADF Library jar file.
- Add reference to resource bundle of ADF Library project
- Import JHeadstart service definitions in main ViewController project

Each of these steps is explained below in more detail.

2.4.1. Remove JhsCommon-beans.xml from the ADF Library jar file

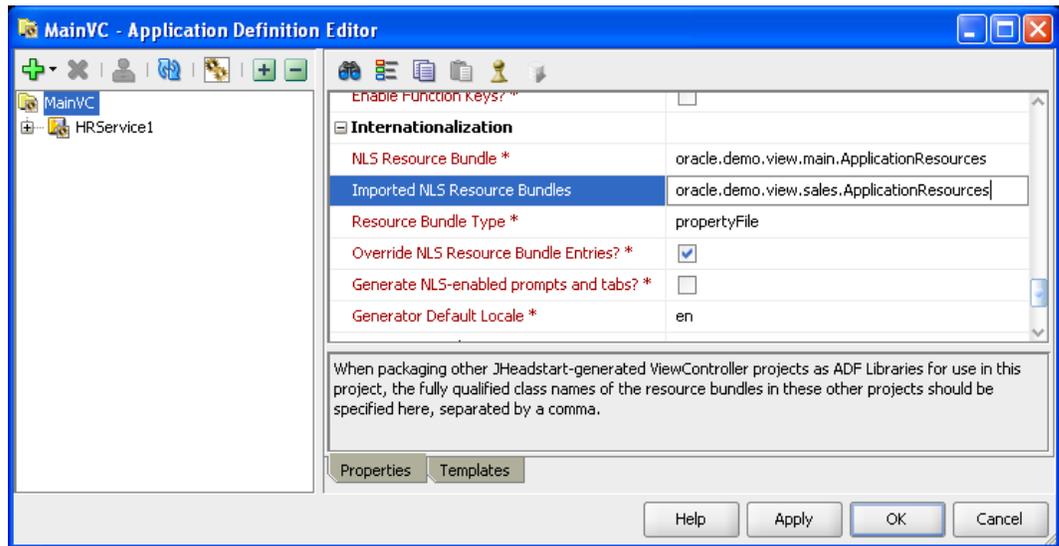
In each ViewController project, JHeadstart will generate an ADFC configuration file named JhsCommon-beans.xml. This file holds a number of application-wide managed bean definitions. The ADF Library created for the ViewController project includes this file, and so does the main ViewController project. If you subsequently run the application from the main ViewController project you will get warnings in the log file like this:

```
<MetadataService$Bootstrap><add> ADFc: /WEB-INF/JhsCommon-beans.xml:  
<MetadataService$Bootstrap><add> ADFc: Duplicate activity 'Home' detected.  
<MetadataService$Bootstrap><add> ADFc: /WEB-INF/JhsCommon-beans.xml:  
<MetadataService$Bootstrap><add> ADFc: Duplicate activity 'CallMenuItem' detected.
```

The application will still run, however, if you want to get rid of these warnings, the ADF library jar file should not contain the JhsCommon-beans.xml. You can either remove this file from the jar file after you created the jar file, or you can remove the file altogether from the ViewController project that is packaged as ADF library. If you remove the file from the project, you will no longer be able to run the application subsystem directly in the supporting ViewController project. Unfortunately, there is currently no option in JDeveloper to specify specific files that should be excluded when deploying to an ADF library.

2.4.2. Add reference to resource bundle of ADF Library project

At runtime, JHeadstart uses one generic managed bean named "nls" to provide translatable resources. Under the covers, this managed bean accesses one or more resource bundles. When you import a ViewController project as ADF Library, the JHeadstart-generated resource bundle of this imported project should be specified in the JHeadstart Application Definition Editor of the main ViewController project, in property **Imported NLS Bundles**, which is available at the application level.



2.4.3. Import JHeadstart service definitions in main ViewController project

The JHeadstart metadata services and groups within the services defined in the Application Definition Editor of supporting ViewController projects are by default not available in the Application Definition Editor of the main project.

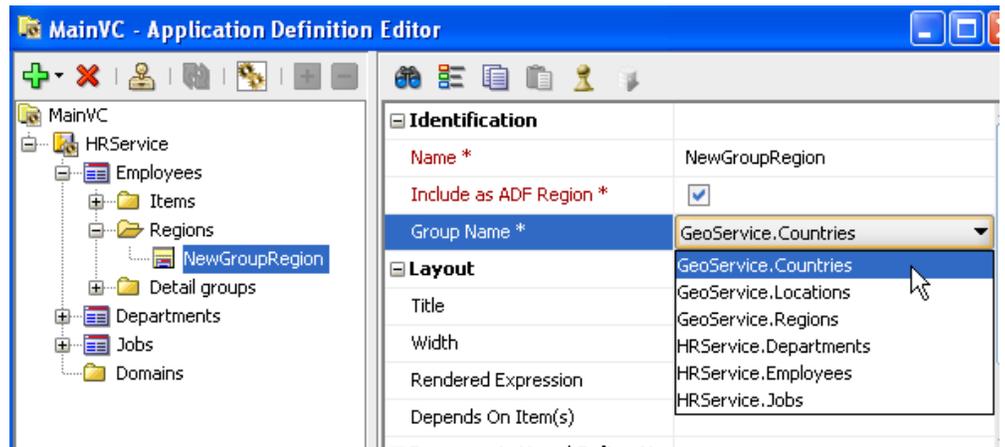
To make this information available in the main project, you can specify so-called imported service definitions in the JHeadstartApplicationDefinition.xml file, as shown below.



You need to add this reference directly in the XML file. Make sure you set the **imported** property to true. The location property references the actual service definition xml file in the other project. This can either be a fully qualified absolute path, or a relative path as shown above.

If you added this imported service definition reference, the following will happen:

- All drop-down list properties in the JHeadstart Application Definition Editor where you can pick a group, will also show the group names of the imported service library. For example, this enables you to reuse LOV groups defined in other projects. You can also define links between groups that are generated into different projects.



- The generated menu model will include top-level menu entries for the imported service definitions.



Note that an imported service definition will NOT show up in the navigator tree within the JHeadstart Application Definition editor.

2.5. 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.



ADF – Enterprise Methodology Group. The ADF – EMG is a group of ADF users, both experts and novice users, both Oracle employees as well as customers and partners that discuss anything beyond the typical “How do I” questions that are more suited for the JDeveloper discussion forum. The ADF-EMG group also maintains web pages around ADF Coding Standards.

2.5.1. 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 for business 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

2.5.2. 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 one 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, HumanResourcesService

This page is intentionally left blank.

Creating ADF Business Components

This chapter provides you with guidelines on creating ADF Business Components. The ADF Business Components. The Fusion Developer's Guide for ADF 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 3 to 12 of the Fusion Developer's Guide for ADF.

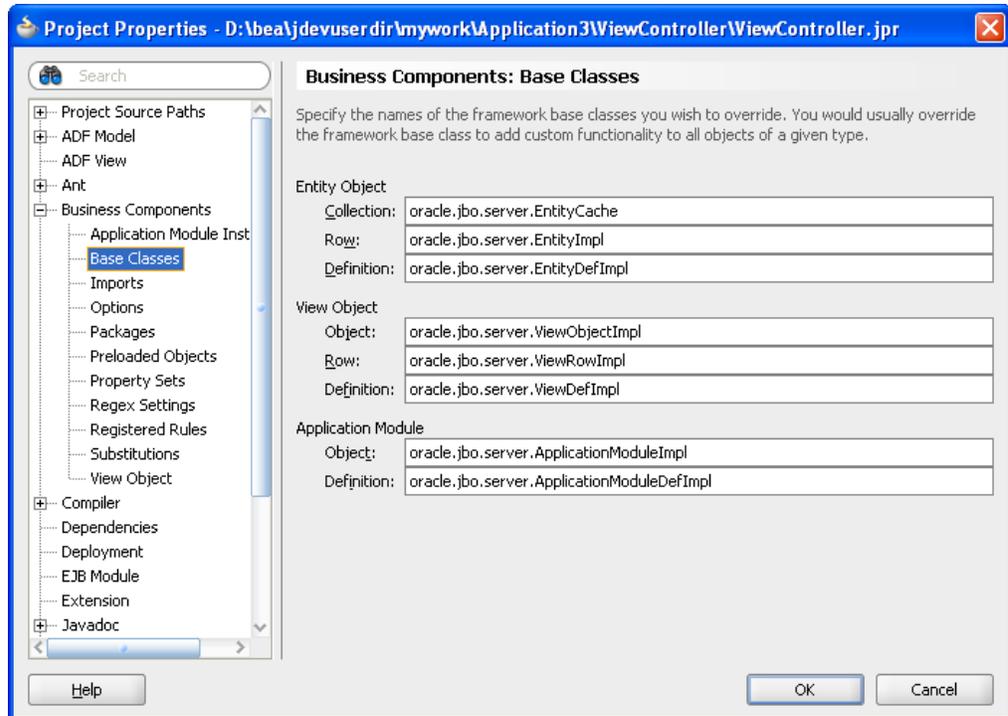


Fusion Developer's Guide for Oracle Application Development Framework., Chapters 3 to 12 describe the basics for creating ADF Business Components. Chapters 42 to 44 contain advanced techniques.

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/toc.htm

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 at global JDeveloper level in menu option Tools – Preferences, or for a specific project in the Project Properties – ADF Business Components Panel.



In the Fusion Developer's Guide for ADF 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

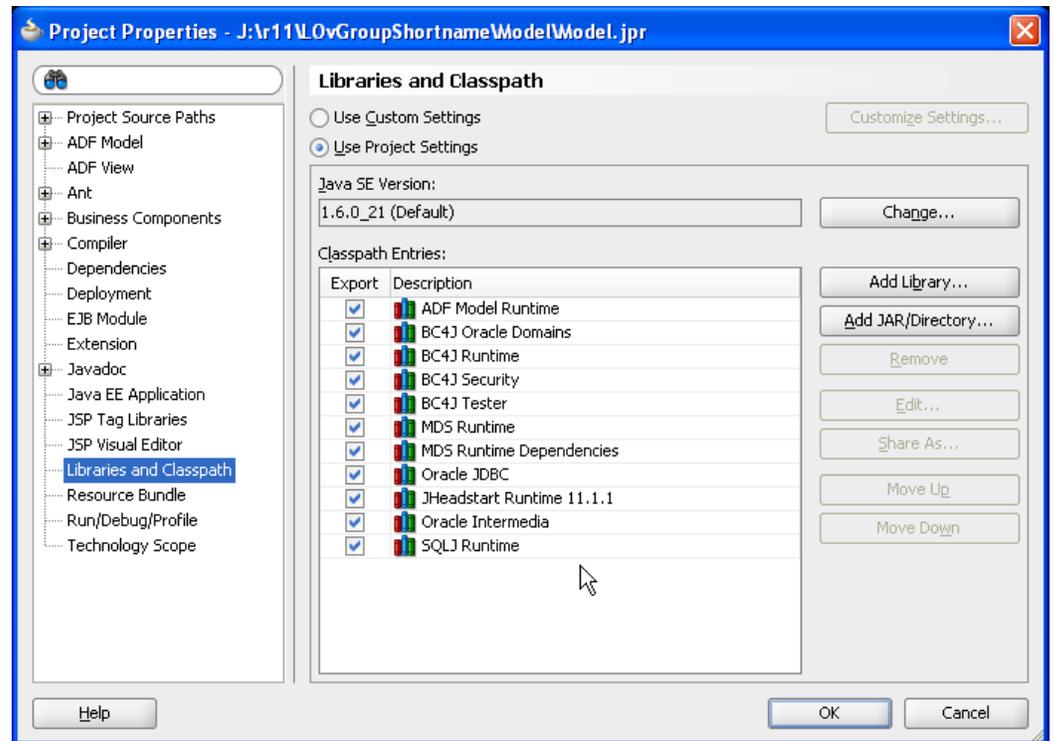
You will be glad you heeded this recommendation. Failure to set up these preferences at the outset can present your team with a substantial inconvenience if you discover mid-project that all of your entity objects, for example, require a new generic feature, augmented built-in feature, or a generic bug 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 section 37.2 of the Fusion Developer's Guide for ADF.

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 its 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 11.1.1 library is added to your project.

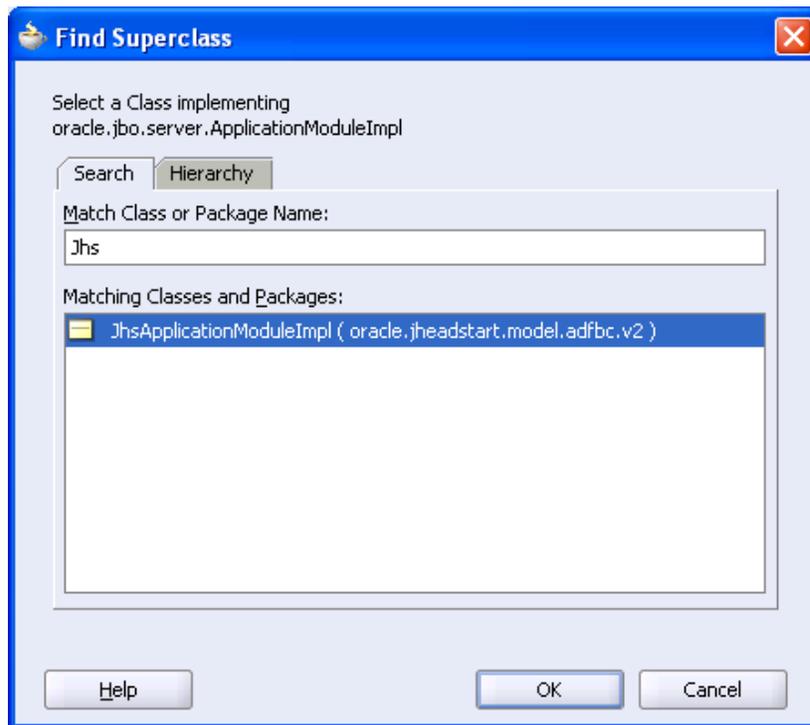


- Go to the Project Properties
- Select category Libraries and Classpath
- Click Add Library
- Under Extension, select JHeadstart Runtime 11.1.1
- Click OK twice
- Save the project

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.

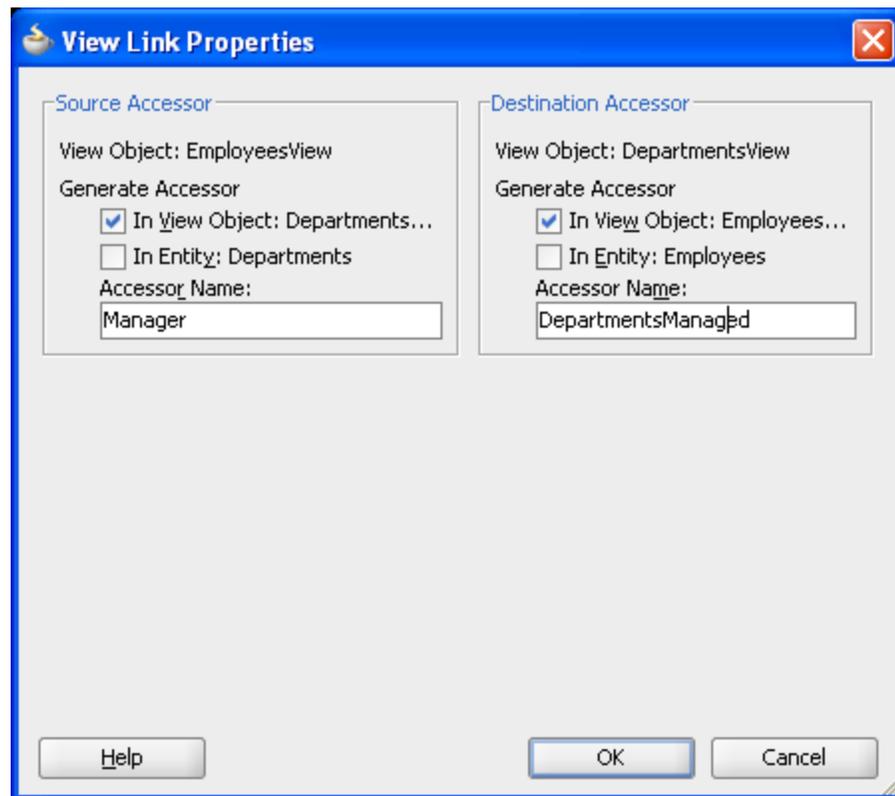
3.2.3. Renaming Entity Objects and Associations

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 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.

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”.

Behavior

Specify the behavioral aspects of this association.

Use Database Key Constraints

Composition Association

Optimize For Database Cascade Delete

Implement Cascade Delete

Cascade Update Key Attributes

Update Top-level History Columns

Lock Level:

None Lock Container Lock Top-level Container

Effective Dated 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 InvalidOwnerException”.

<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:

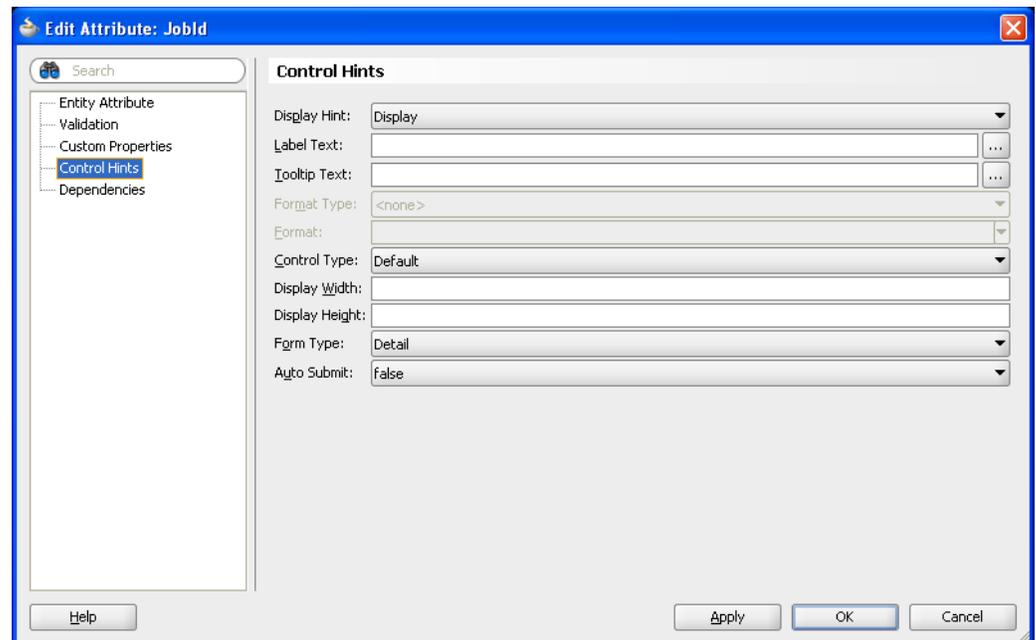
- **Mandatory:** when this checkbox is checked, the item that is based on this attribute will be generated with an asterisk in front of the label, and a JavaScript alert will be displayed when you submit the page when the item is still empty.
- **Queryable:** when checked, the item created for this attribute 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 Model project.



Internationalizing Control Hints. Explained in ADF Developers Guide. http://docs.oracle.com/cd/E24382_01/web.1112/e16182/bcentities.htm#sm0140

By default, JHeadstart will use an EL expression to reference the UI Hints set in the model. You can also configure JHeadstart to copy the UI hints and not use an EL expression to reference it. See chapter 11 “Internationalization” for more information.

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 Fusion Developer’s Guide has extensive information on implementing business rules, see the references below. In addition, 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. While the paper has been written for release 10.1.3, the main concepts are still applicable.



Web Reference: Fusion Developer's Guide for Oracle Application Development Framework, chapter 7: Defining Validation and Business Rules Declaratively

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/bcvalidation.htm#m0231



Web Reference: Fusion Developer's Guide for Oracle Application Development Framework, chapter 7: Defining Validation and Business Rules Programatically

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/bcrules.htm#CIHB BDDB



Implementing Business Rules in ADF BC. White paper on OTN.

<http://www.oracle.com/technology/products/jdev/collateral/papers/10131/businessrulesinadfbtechnicalwp.pdf>

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, the following conditions must be met:

- Each View Object must have at least one key attribute
- **Primary keys must be pre-populated for new records.**
- The key attribute(s) should be non-updateable

The second condition has rather big implications: your datamodel must use surrogate primary keys so you can prepopulate them in the Create method of your entity objects (JDeveloper bug 6894412). If you are building an ADF application against an existing datamodel with meaningful primary key attributes that are populated by the end user, you need to add an additional ID column to your database tables, and mark the attribute for this column as key attribute. Note that you do NOT need to change the existing database constraint definitions although it is good practice to define a unique key on this new ID column.

A view Object key that is updateable will 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 will not work correctly anymore.

3.3.3.1. 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 37.3.2 of the ADF Developer's Guide:

http://download.oracle.com/docs/cd/E17904_01/web.1111/b31974/bcadvgen.htm#sm0296

3.3.4. Setting View Object Control Hints

You can specify Control Hints for an attribute in a View Object.

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](#) for 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 requiered 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.



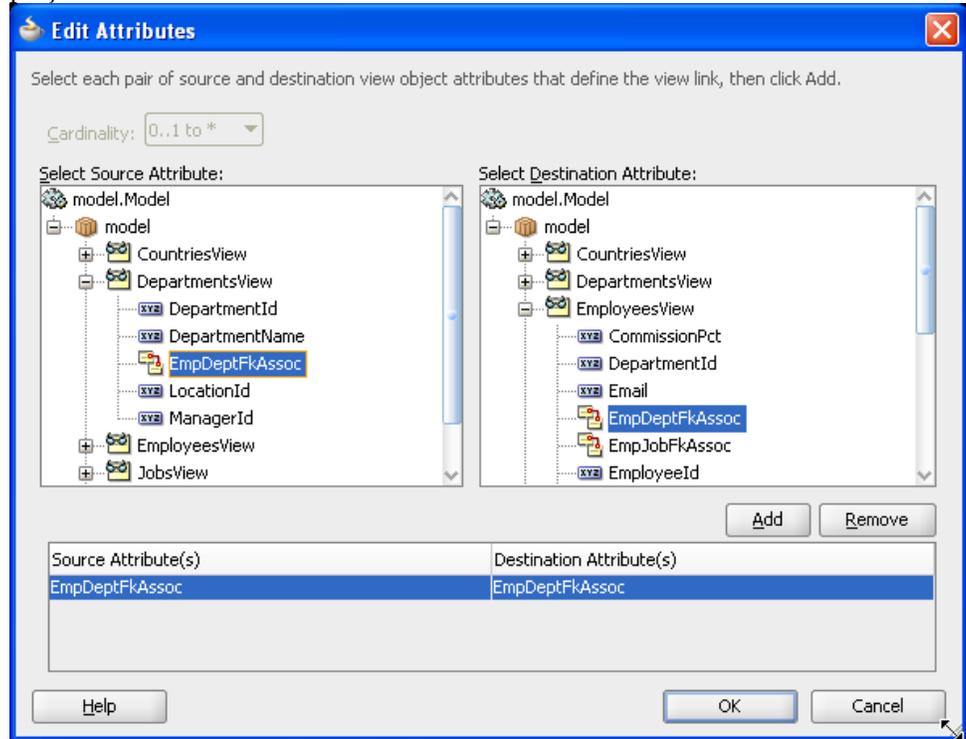
Reference: Fusion Developer’s Guide for Oracle Application Development Framework, section 5.14: Adding Calculated and Transient Attributes to an Entity-Based View Object.
http://docs.oracle.com/cd/E24382_01/web.1112/e16182/bcquerying.htm#CHDHJHBI

3.3.7. Setting Up Master-Detail Synchronization

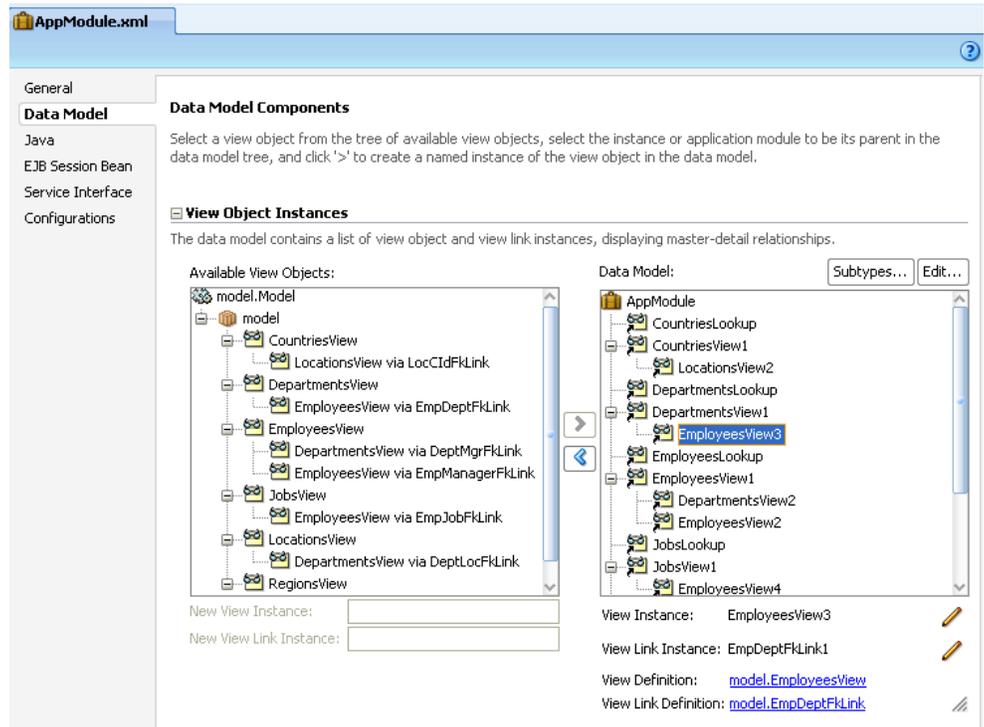
JHeadstart is capable of generating parent-child or master-detail layouts. For example you want to show a department with all the employees in that departments as detail.

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 'Run...'. 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.

This page is intentionally left blank.

Using JHeadstart

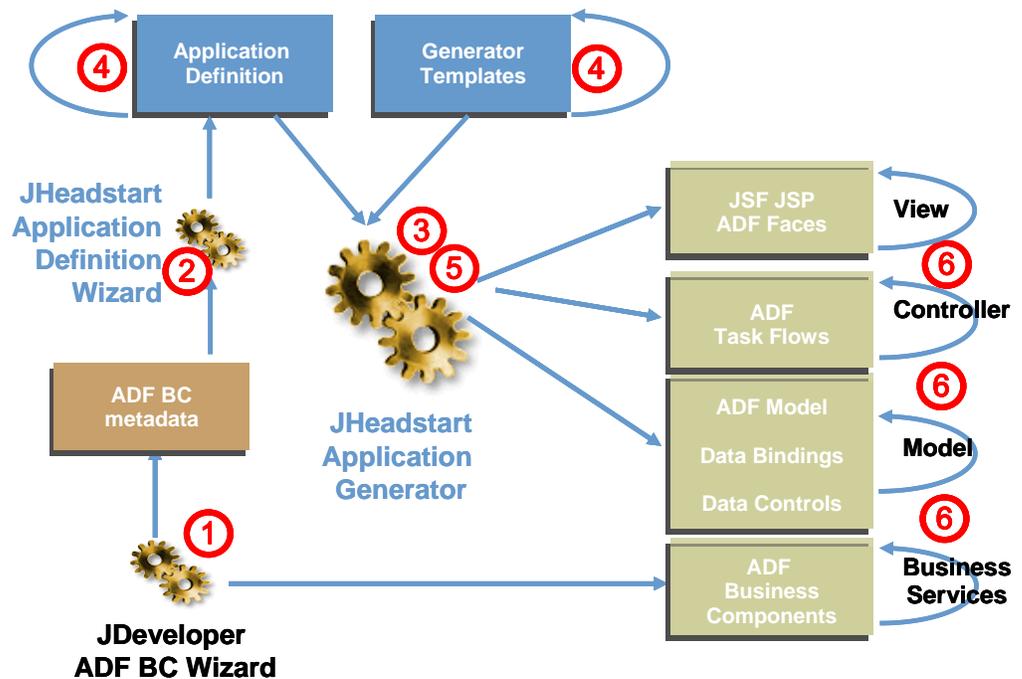
This chapter provides information on how to use JHeadstart in general. The following JHeadstart in general. The following topics are covered:

- Understanding the Generator Architecture
- Using the JHeadstart Enable Project Wizard
- Using the Create New Service Definition Wizard
- Using the Application Definition Editor
- Running the JHeadstart Application Generator
- Running the Generated Application
- 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 (Rich Client).
- Controller: JSF/ADF Controller.

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.
- JSF fragments for each displayed region.
- ADF Task Flow configuration files for each group (of pages).
- Page Definitions (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, results in a fully functional web application.

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

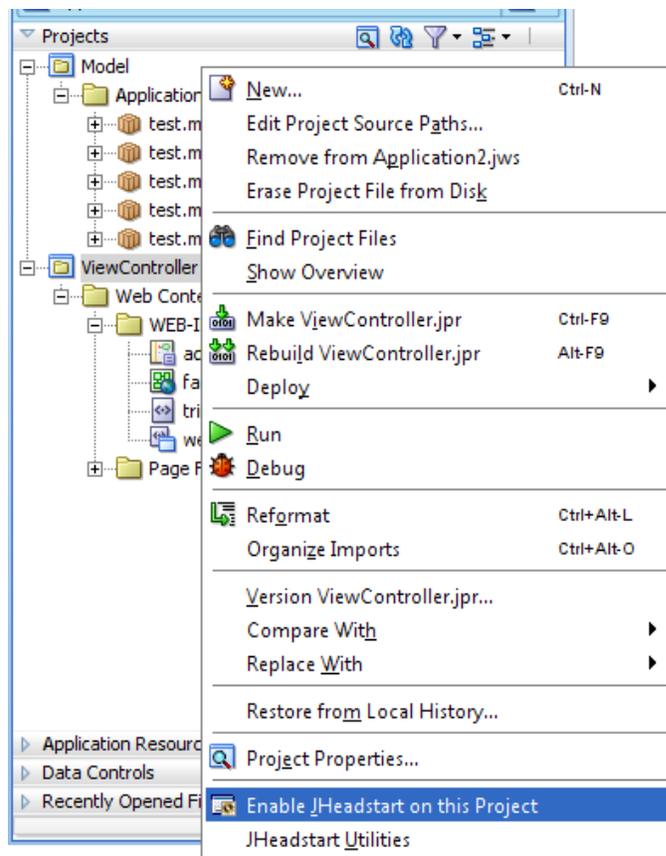
4.2. Using the JHeadstart Enable Project Wizard

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.

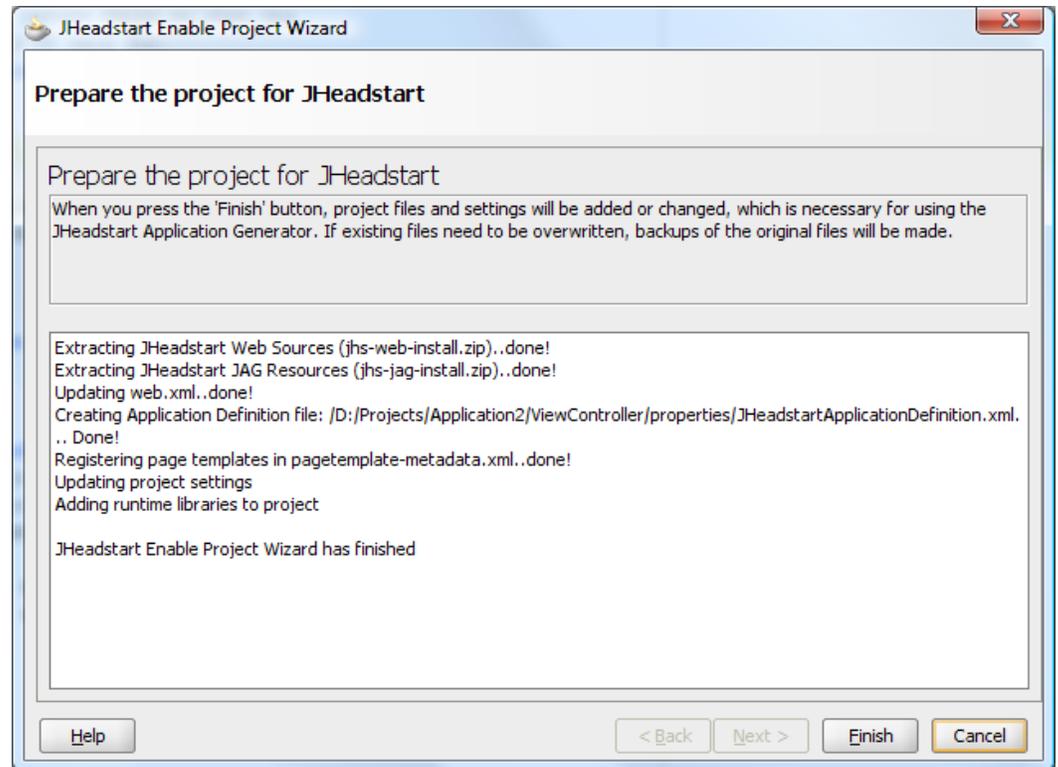
4.2.1. Enabling JHeadstart on a new project

Enabling JHeadstart on a project is a simple operation that you can perform by clicking the alternative mouse-button 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.

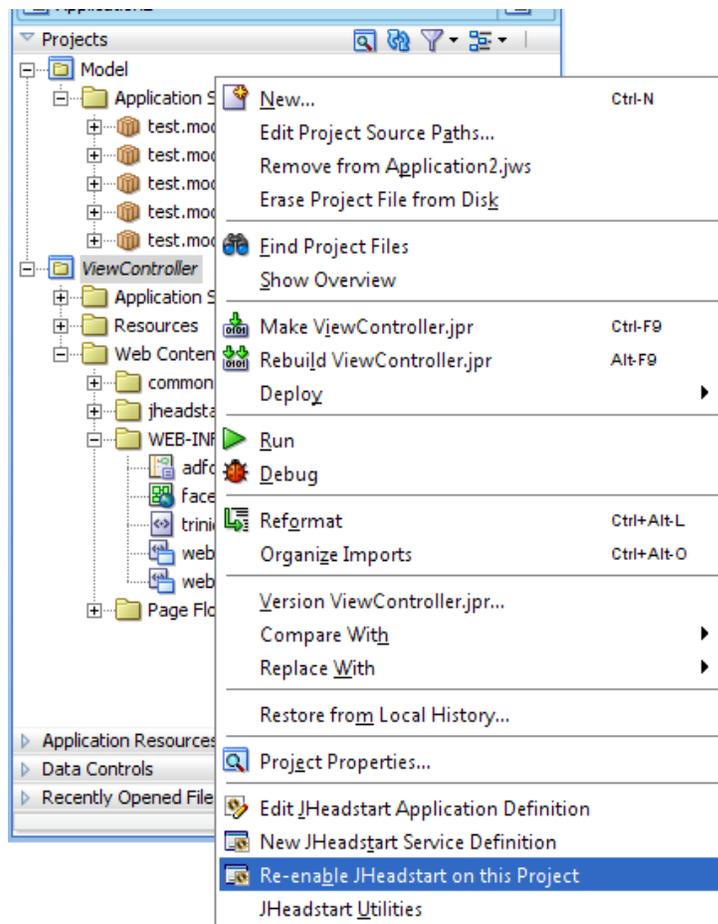


This is done for all other files, such as Tag Libraries and JHeadstart-specific files. It is unlikely that you made manual changes to these files, so normally you would choose 'Overwrite All', but you can make your choice to overwrite, backup or ignore on a per-file basis if you want.

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, click on the project with the alternative mouse-button and choose 'Re-enable JHeadstart on this Project'.



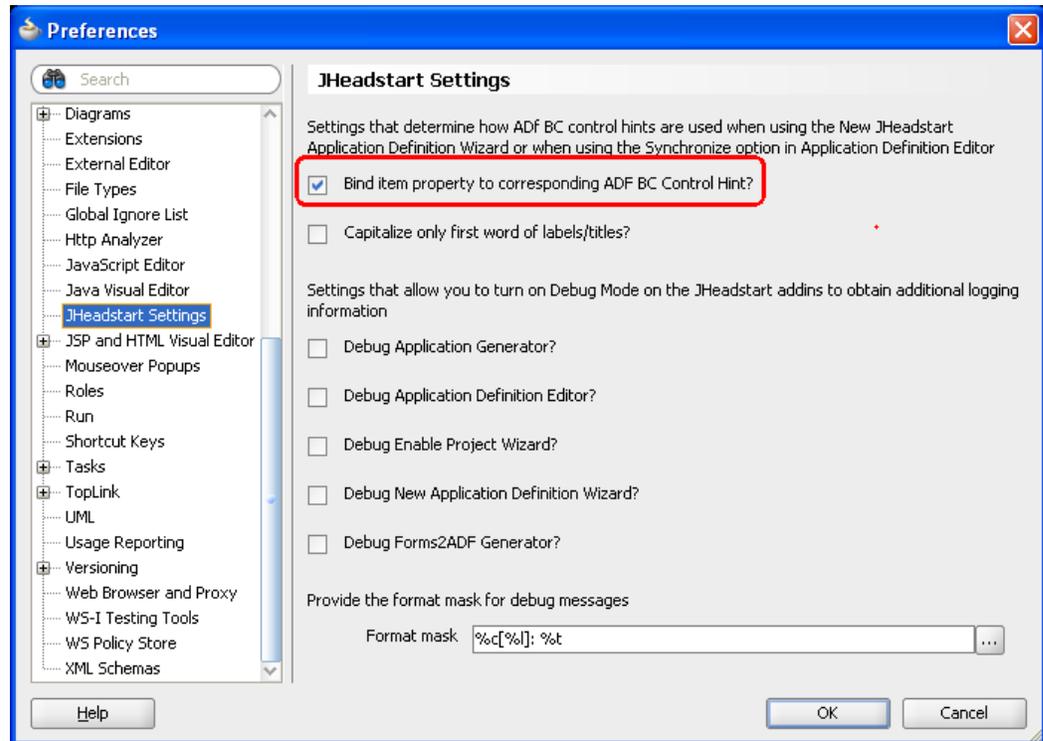


Attention: As you can see, because JHeadstart was already enabled on this project, you can choose the menu options 'Edit JHeadstart Application Definition' and 'New JHeadstart Service Definition', and the menu option to launch the JHeadstart Enable Project Wizard was renamed to 'Re-enable JHeadstart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

4.3. Using the Create New Service Definition Wizard

After enabling JHeadstart, you will typically create a JHeadstart Service Definition XML file. You can create a new Service Definition by right-clicking the ViewController project and choosing 'New JHeadstart Service Definition'. You will be presented with a 5 step wizard.

Before running this wizard, you should inspect the JHeadstart Preferences Settings, as they affect how the new service definition is created. You access the JHeadstart Preferences Settings through the Tools -> Preferences menu in JDeveloper.



There are two preferences that are relevant when running the New Service Definition Wizard:

- **Bind Item property to corresponding ADF BC Control Hint?:** when checked the **Prompt in Form, Display Width, Display Height and Hint (Tooltip)** properties of items created in the wizard will **refer** to the corresponding Control Hint set on the underlying View Object attribute. When not checked, the value of this Control Hint is **copied** to the item property of the created JHeadstart item.
- **Capitalize only first word of labels/titles:** This property is only applicable when the **Bind Item property to corresponding ADF BC Control Hint?** preference is unchecked. If so, this preference determines whether a View Object attribute name like LastName will result in a default prompt of "Last Name" or "Last name".

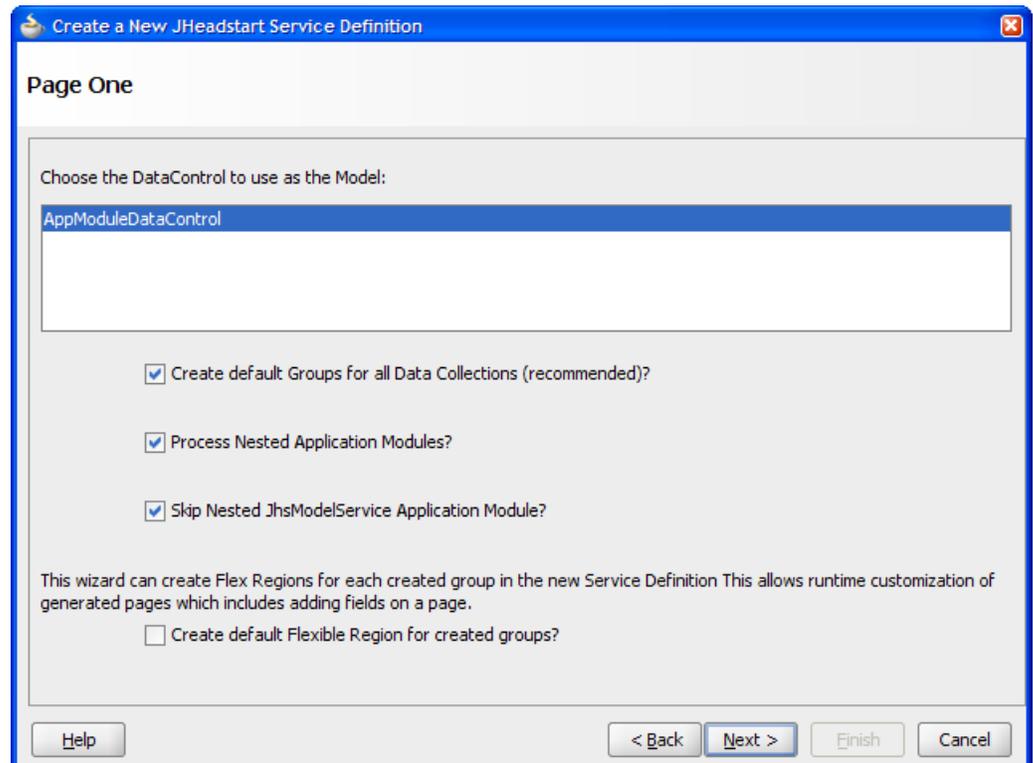


Attention: The following section assumes you are familiar with the concepts that are introduced in this wizard (such as List of Values, search and layout styles, etcetera).

Fortunately, the wizard sets the most commonly used values as default. If you are not sure what option to choose, leave the default as is.

The first screen of the Wizard (step 0) is an introduction, simply click Next to get to step 1.

4.3.1. Step 1: Choose Data Control



The list on the top of this page lists all the DataControls (i.e. Application Modules) that are found in all the Model projects within your workspace. For example: if you have 3 Model projects with 2 application modules each, you will see 6 entries here.

A Service Definition always corresponds to exactly one Application Module, so you can not select multiple DataControls from the list.

The following options are available on this screen:

- **Create default groups:** when unchecked, the new Service Definition will not contain **any** predefined groups. If checked, it will create one group for every root View Object usage, plus detail groups for every nested VO usage.
- **Process Nested Application Modules:** when checked, JHeadstart will also create (detail)Groups for all VO usages within nested application modules.
- **Skip Nested JhsModelService Application Module:** JHeadstart will nest the JhsModelService Application Module inside your own Application Module, to use special JHeadstart features such as dynamic menu, translations etcetera. Normally, you will not want to create groups for that application module, so leave this checkbox ticked.
- **Create Default Flexible Region:** If you want to use 'flexible regions' throughout your service by default then check this box. Of course it is also possible to add flex regions later on. See [section 12.2](#) about Flex Regions for more information.

4.3.2. Step 2: Choose Basic Settings

Page Two

Specify the name of the Service/Application.

Service Name

Specify the name of the Service Definition

File Name

Specify the default layoutStyle for Parent Groups

Parent Layout Style

Specify the default layoutStyle for Child Groups

Child Layout Style

Help < Back Next > Finish Cancel

Here you can specify the name of the Service, which will appear in some filenames and in the list of 'Modules' in your runtime application.

Also the basic layout styles for 'Parent Groups' (base groups directly connected to the service) and 'Child Groups' (detail groups that are connected to another group) can be specified.

4.3.3. Step 3: Choose List of Values Options

Page Three

JHeadstart offers a wide variety of Lists of Values. If you choose on of the ADF options, JHeadstart will automatically generate LOV definitions on the appropriate view object attributes in your model. See the JHeadstart Developer's Guide for more information.

List Of Values Type

Where do you want the VO-instances for the lookups to be created? A shared application module is more memory-efficient, but should only be used when your list content does not have to change during a session.

Use existing application module (instance)

Create new shared application module

Create read-only copies of updateable view objects in the following package:

When using dropdowns, you can specify the label that will be used to enter an 'empty' value.

Unselected label

Help < Back Next > Finish Cancel

Here you can choose various options for the List of Values components (see [section 6.7](#) for more information on Lists of Values).

The following options are available:

- **List of Values type:** Here you can choose between the various ADF model types (starting with ADF-) and the JHeadstart types (starting with JHS-). Since you have to choose one setting for all Lists of Values in your application, you should try to choose the setting that will occur most often.
- **Use existing application modules / Create new shared application module:** These options are only editable when you have selected an ADF Model type LOV.
Choose the existing application module (base module) if the LOV components in your application are not only showing “static” content. For example: if your application needs to show newly added Departments in a dropdown list, you should choose the base module.
However, the ‘Shared Application Module’ approach is more memory- and query efficient. So if you show only (fairly) static data in LOV components, choose either a new Shared Application Module or an existing Shared Application Module. See the reference below for more information on Shared Application Modules.
- **Create read-only copies:** This will create a *copy* of your lookup View Object in the package specified *without* using any Entity Objects. In other words, a read-only View Object will be created using the SQL statement of the updateable View Object. Read-only View Objects are a bit faster compared to updateable View Objects and take fewer resources. They have the same drawback as Shared

Application Modules however: so choose this only when you have LOV components with (reasonably) static data.

- **Unselected label:** For dropdowns, it can be possible to select a 'NULL' or empty value. This field specifies the label that will be used for that NULL value.



Fusion Dev Guide, chapter 10 “Sharing Application Module View Instances”. Contains more information on Shared Application Modules.

http://download.oracle.com/docs/cd/E17904_01/web.1111/b31974/bclookups.htm#BABEIAIB

4.3.3.1. Choosing a List of Values Type

There are various options you can choose for as a List of Values type. You should be careful here in what option to choose, depending on the number of rows that will be displayed per LOV.

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: 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, then choose a separate (popup) LOV style. This includes ADF-inputTextLov, ADF-comboBoxLov and JHS-LOV.

4.3.3.2. Understanding What JHeadstart Generates

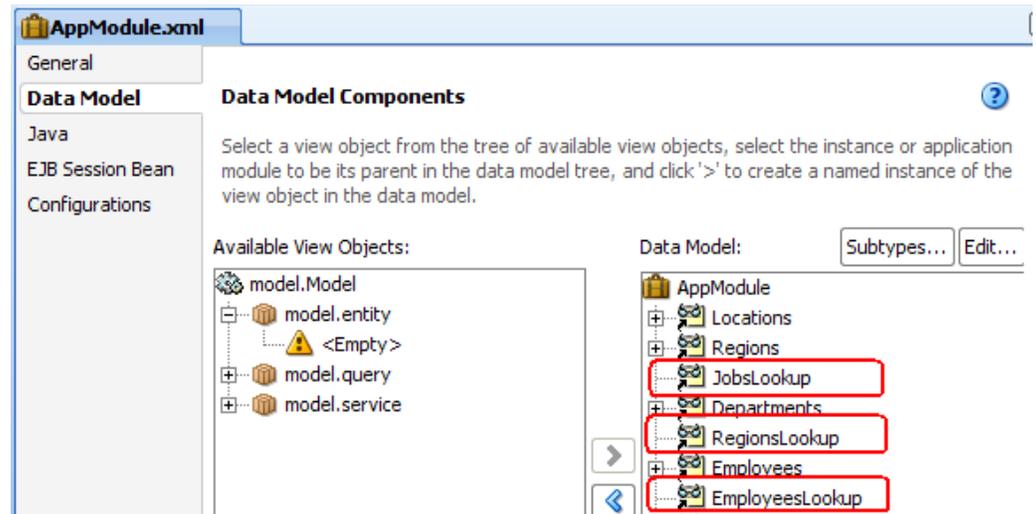
There are three basic scenarios for LOV generation. Each of these scenarios gives a different result in terms of files that are generated, view object instances added, JHeadstart groups that are added, etcetera.

Scenario 1: JHeadstart LOV/dropdown

When you choose a JHeadstart LOV/dropdown in the wizard, the following objects will be created:

Lookup View Object Usages

For each View Link, JHeadstart will generate 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 after the wizard has finished:



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.

LOV Groups or Dynamic Domains

If you choose “JHS-dropdown” as LOV type, JHeadstart will create a dynamic domain in the Service Definition File and set the display type of the item to dropDownList. The domain of the item will point to the new dynamic domain.

If you choose a JHeadstart LOV, JHeadstart will create a new LOV group (with the same name as the new Lookup View Usage). Furthermore, it will add a List of Value to the LOV item that will point to the new LOV group.

Scenario 2: ADF Model LOV without Shared Application Module

In this case, JHeadstart will **not** create any ‘Lookup’ view usages in the base application module.

Instead, JHeadstart will create a View Accessor and a List of Values definition for every View Object attribute that has a View Link. After the wizard has finished, the LOV can be inspected by looking at the attributes of a View Object:

Name	Type	Column	Info
DepartmentId	Number	DEPARTMENT_ID (De...	
DepartmentName	String	DEPARTMENT_NAME ...	
ManagerId	Number	MANAGER_ID (Depart...	
LocationId	Number	LOCATION_ID (Depar...	
SpecialInd	Number	SPECIAL_IND (Depart...	

Custom Properties: ManagerId

List of Values: ManagerId

Enable this attribute to display a list of values to use in the user interface.

Lists of Values:

Name	List Data Source	List Attribute
LOV_ManagerId	Departments_EmployeesLo...	EmployeeId

When we click the 'edit' icon on the List of Values, we see the following:

As you can see, JHeadstart has defined a complete List of Values definition in our attribute, including information on which attributes should be copied from the LOV to the base page.

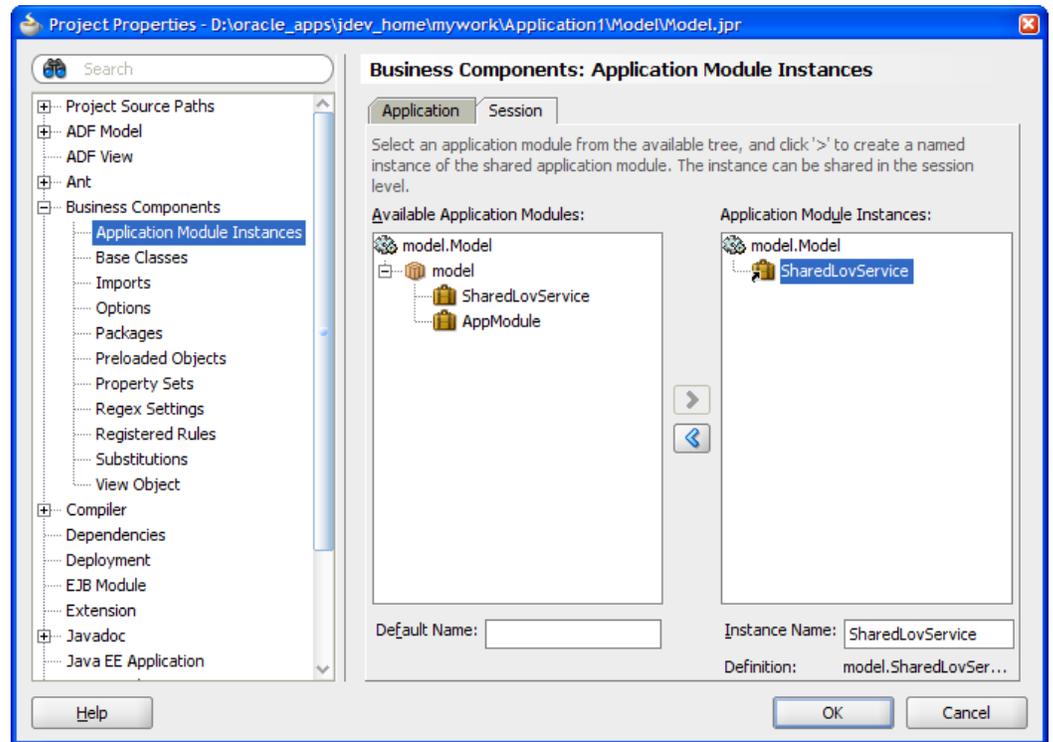
The new Service Definition file will not contain any LOV groups or dynamic domains. Instead, it will simply set the display type of the LOV item to the appropriate value (e.g. "model-inputTextLov"). This is enough information for JHeadstart to generate the page.

Scenario 3: ADF Model LOV with Shared Application Module

This scenario looks a lot like the second scenario. However, something extra will be done:

Creating a New Shared Application Module

When a New Shared Application Module needs to be created, first a new (empty) Application Module will be created in the package of your choice. It will also be added as a *session instance* to your Business Components project. This means one instance of this Application Module will be created per session. You can see this when right-clicking your Model project, choose Properties and go to the Business Components > Application Module Instances tab:

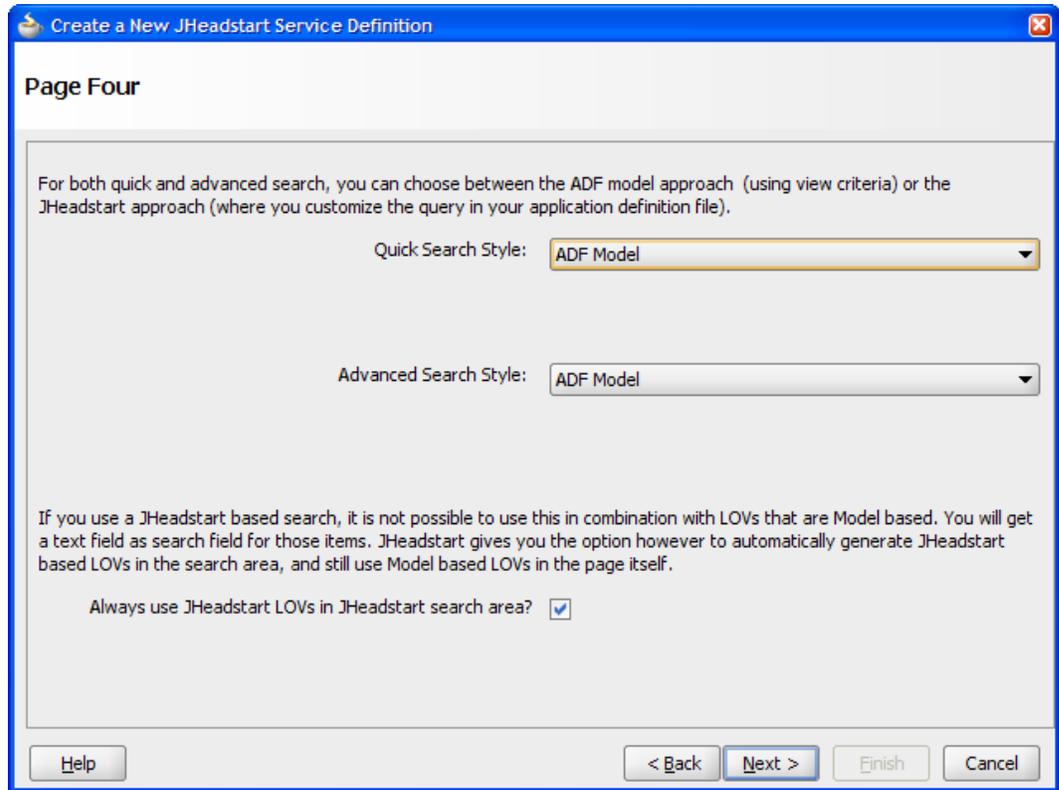


Here you see that the new 'SharedLovService' has been added as a session instance.

Adding Lookup View Usages

When JHeadstart generates a new Shared Application Module, it will also add a Lookup View Usage to it for every LOV that is generated. This happens in the same way as for JHeadstart dropdowns/LOVs; except they are not added to the base App Module but to the Shared Application Module.

4.3.4. Step 4: Choosing Search Style



Here you can choose the different default search styles for quick and advanced search, as described in [section 7.2](#).

In general, we recommend using the ADF Model approach, unless you have requirements that cannot be implemented using the ADF Model search. Such requirements include:

- Specific layout requirements for the search area, for example, organizing the search items in groups with headers, or multiple tabs.
- Read-only descriptor items that should be populated by an LOV for a search item
- Usage of JHeadstart List of Values in the search area.

The compatibility matrix in [section 6.7.1](#) gives you more information on the issue of LOV and search combinations that are 'invalid'. The rule of thumb is quite simple: either choose ADF Model LOV and the ADF Model search together, or choose the JHeadstart LOV and JHeadstart search together. Do not try to mix the two approaches.

However, if you (really) need to use the JHeadstart search and the ADF Model LOVs together, this wizard gives you a bit of extra help. If you tick the checkbox on the bottom of the screen, JHeadstart will create two items for each attribute with an LOV:

1. An **ADF-Model** based LOV item that will be displayed in your form/table. It will **not** be displayed in your JHeadstart search area (since that is not supported).

2. A **JHeadstart** based LOV item that is **not** shown in the form/table layout, but is only shown in the JHeadstart search area.

Both items are based on the same attribute in the View Object and have the same settings (including label).

4.4. Using the Application Definition Editor

JHeadstart uses an Application Definition and (multiple) Service Definitions to define the structure of your application. These definitions are stored in separate xml files. They identify which pages you need, how you want these pages related, their layout styles, what information sources they are based on, and so on. The Application Definition defines the global settings for your entire application. Each Service in the Application Definition will be generated with its own menu structure (e.g. tabs). Each (top level) group in the service corresponds to one item in this menu (e.g. a tab).

One Application Definition can contain multiple services. To create another service, run the 'New JHeadstart Service Definition' again.



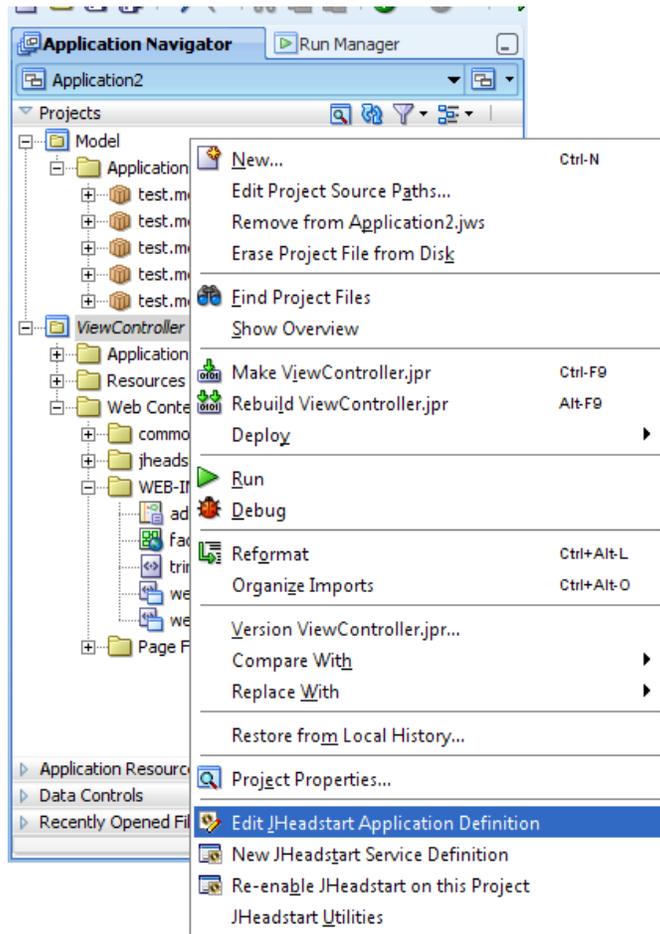
Reference: How to create a Service Definition is described in section 4.3, *Using the Create New Service Definition Wizard*. This section only discusses how to create new groups, and how to modify and remove existing groups.

4.4.1. Maintaining the Application Definition

The JHeadstart Application Definition Editor helps you to maintain the Application and Service Definitions without having to write and edit the XML files 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

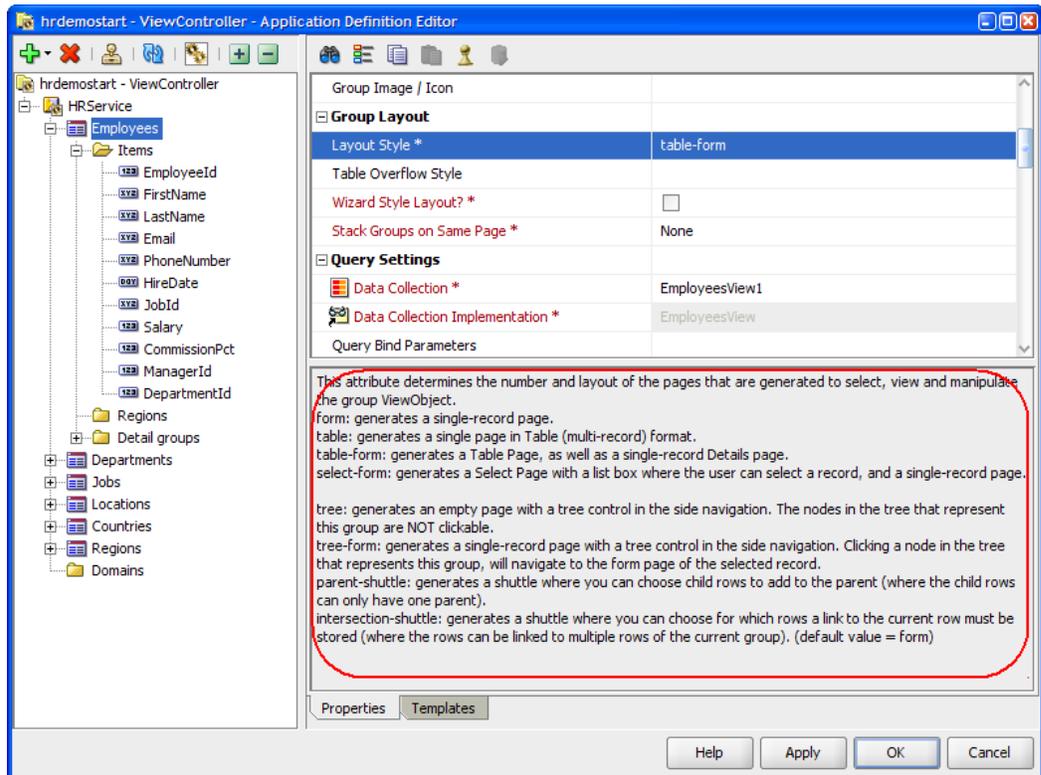
To be able to start the editor you must have enabled JHeadstart on the project. Press the alternative mouse-button on the Project and click 'Edit JHeadstart Application Definition' to open the editor:



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.

Validation	
Required?	<input type="text" value="#{bindings.\$BINDING_NAME\$.mandatory}"/>
Validator Binding	
Regular Expression	<input type="checkbox"/> true <input checked="" type="checkbox"/> false

Determines whether a field should be required in the page.
 You can use an EL expression in this property to conditionally make the item required.
 You can use the following key words in the JSF expression, that will be replaced by the JAG:

- `#{BINDING_NAME}` the name of the value binding created for this item in the page definition.
- `#{GROUP_NAME}` the name of the group the item is in.
- `#{PARENT_GROUP_NAME}` the name of the parent group of the group the item is in.
- `#{DEPENDS_ON_ITEM_VALUE}` JSF Expression that returns the value of the item specified in the "Depends on Item" property.

The actual expression is different in table and form layout. By using this keyword, JHeadstart will use the appropriate syntax in table and form layout.
 (default value = false)

This is an editable dropdown list, also called a combo box. If you type a value manually, you need to use the Enter key to confirm your changes.

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.



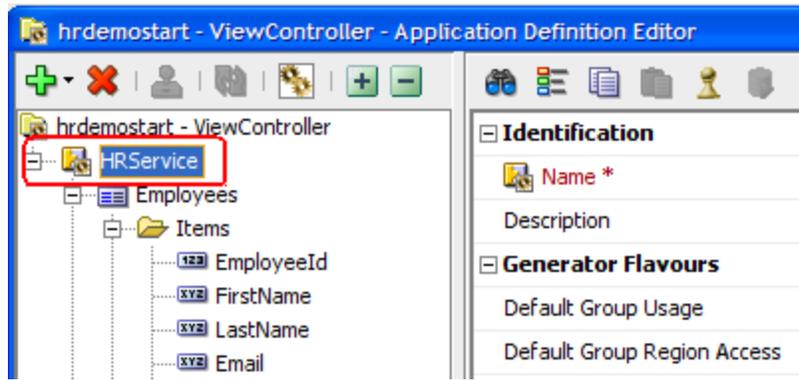
Warning: If you type in a value in a combo box, and then press Tab or click the mouse in a different cell, your typed-in value will be lost. Use Enter to confirm your changes.

4.4.2. Application

The Application Definition (the top node in the Application Definition Editor) is used for global settings that affect the whole application.

4.4.3. 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.

4.4.4. 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.

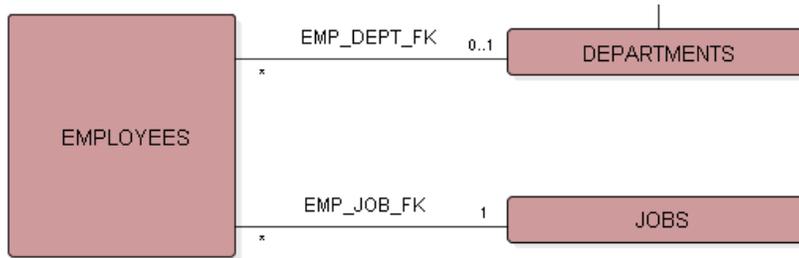
4.4.5. Items

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.

4.4.6. Lists of Values

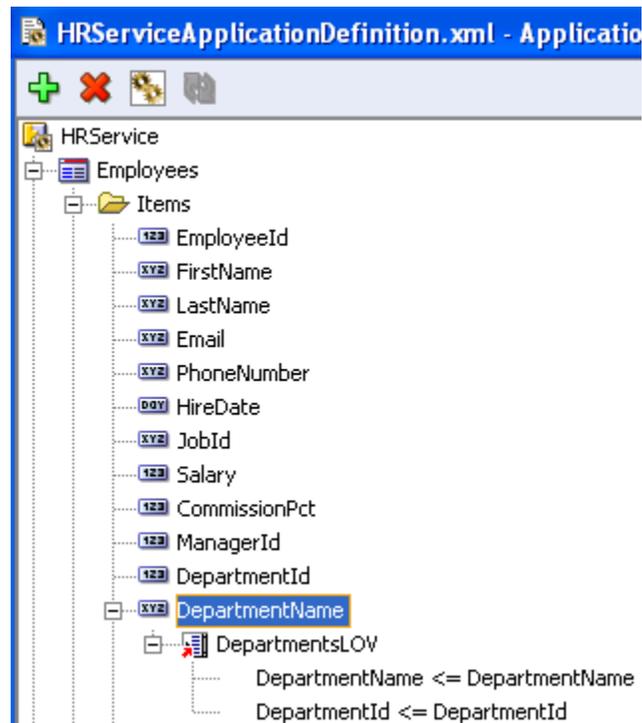
A List of Values (LOV) construction in an Application Definition links an item to a group of type LOV, and specifies which items in the LOV group are to be mapped to which items in the base group. This is required if in the generated application you want to populate the item using a List of Values popup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

4.4.7. 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. 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 **Group Region** can be used to create a dedicated region for a detail group (nested group) that has the **Same Page** property checked. This way one has more control over the placement of the detail group on the page. Alternatively, when the **Include as ADF Region** checkbox is checked on the **Group Region**, you can reference another top-level group that will be included as a reusable region in the page.

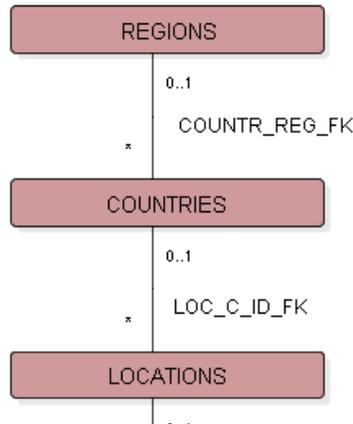
A **Region Container** is, as the name says, a container for regions. The Regions folder is an example of a Region Container.

For examples of using regions, see chapter 5, section Controlling Page Layout Using Region Containers, Item and Group Regions.

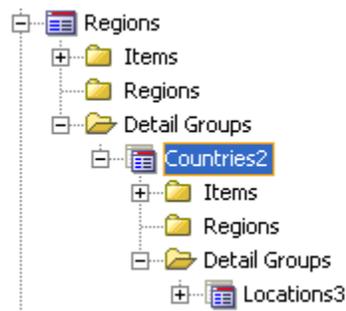
4.4.8. Detail Groups

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:



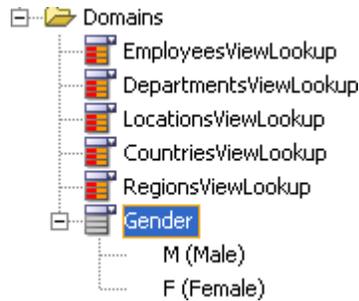
Depending on the layout options you choose, you can display a master group and its detail(s) on different pages or on a single page.

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

4.4.9. 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.

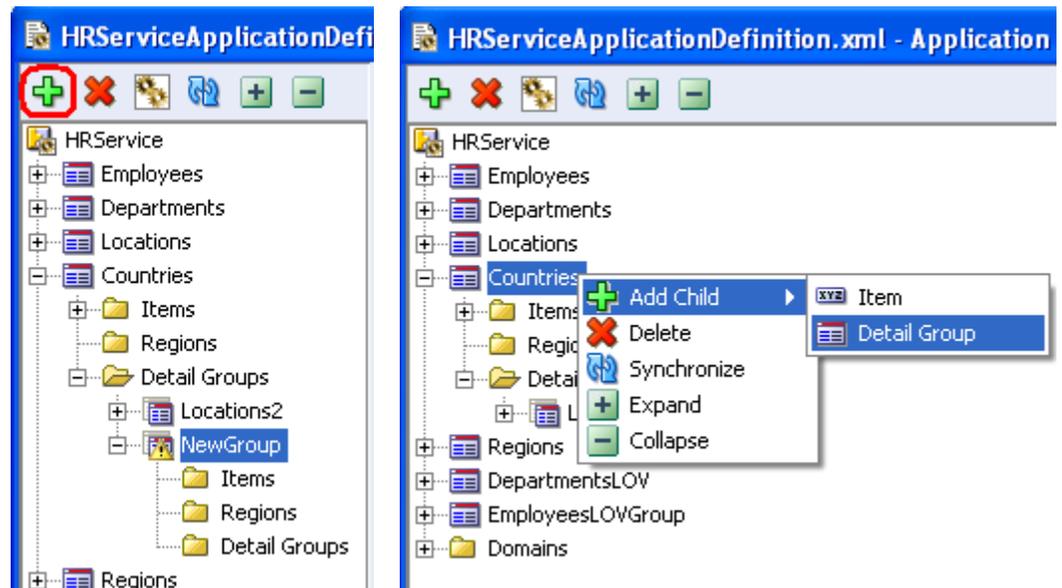


A dynamic domain is a domain based on an ADF View Object and therefore based on a query. See for example the EmployeesViewLookup domain in the screen shot.

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

4.4.10. 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.

4.4.10.1. Moving objects

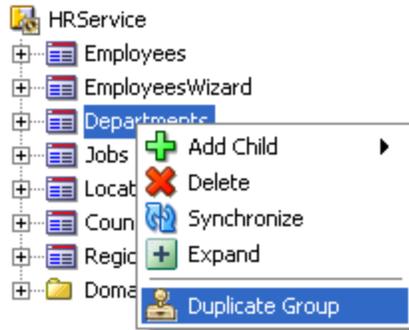
You can move an object in the Application Definition to a different position under the same parent by dragging and dropping the object. This way you can influence the order in which for example level 1 menu tabs or fields in a form layout are generated.

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.

4.4.10.2. 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>".

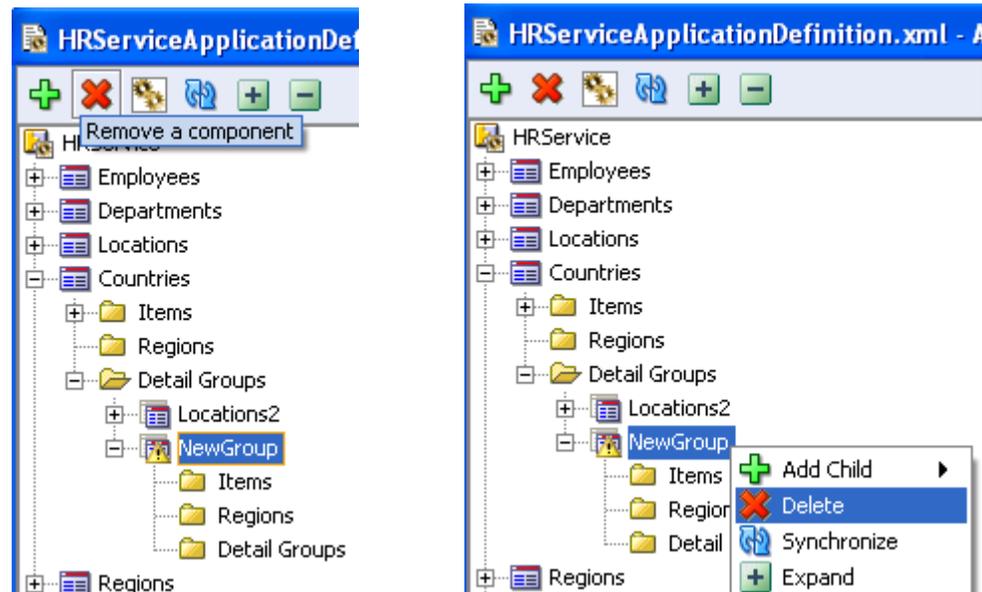
2. You can also copy an object in the Application Definition Navigator by dragging the object while holding the Ctrl key. The cursor will be decorated with a plus (+) in a box to indicate that the dragged object will be copied instead of moved.

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.



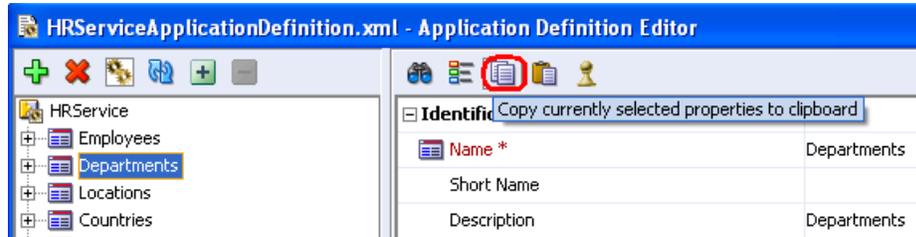
4.4.10.3. Deleting objects

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:

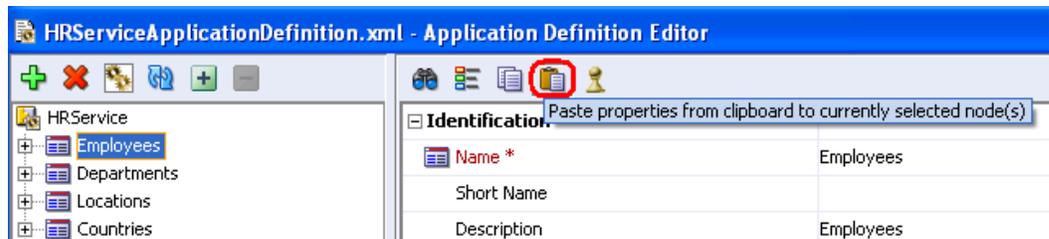


4.4.10.4. Using the clipboard to copy and paste multiple properties

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)':



4.4.11. 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.



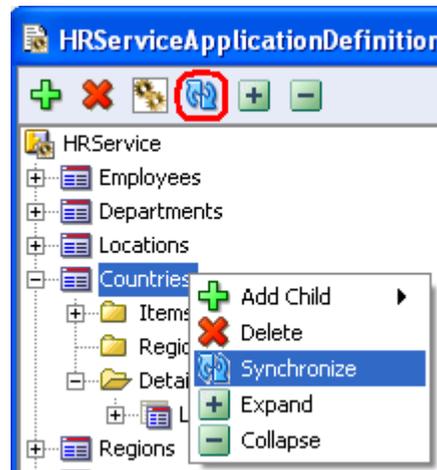
You can switch to expert mode by clicking the icon on the upper right of the editor, as highlighted in the screen shot above. In expert mode you can see all properties, regardless if they are applicable or not.



4.4.12. Synchronize Group with Underlying View Object

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.



The synchronize action will do the following:

- It will add new items in the group for any new attributes in the view object. If you manually create a group in the JHeadstart Application Definition Editor, you can use this option to quickly create all the databound items.
- Existing items that are based on an attribute that no longer exists in the view object are changed into an unbound item.
- The view attribute control hint **Display Hint** will be set to "Display" if the item is displayed in the generated page. (These settings must be in synch; otherwise you will get a PropertyNotFound exception at runtime).
- The view attribute control hint **Label Text** will be set to the value of the **Prompt in Form** property of the item, if the prompt is a literal string value. If JHeadstart Preferences Setting **Bind item property to corresponding ADF BC Control Hint?** is checked, the **Display in Form** item property will be set to the EL expression that references this control hint: `#{SHINTS$.label}`. (The "\$HINTS\$" token will be replaced with the correct EL expression during generation).
- The view attribute control hint **Display Height** will be set to the value of the **Display Height** property of the item, if the property is a literal string value. If JHeadstart Preferences Setting **Bind item property to corresponding ADF BC Control Hint?** is checked, the **Display Height** item property will be set to the EL expression that references this control hint: `#{SHINTS$.displayHeight}`.
- The view attribute control hint **Display Width** will be set to the value of the **Display Width** property of the item, if the property is a literal string value. If JHeadstart Preferences Setting **Bind item property to corresponding ADF BC Control Hint?** is checked, the **Display Width** item property will be set to the EL expression that references this control hint: `#{SHINTS$.displayWidth}`.
- The view attribute control hint **Tooltip Text** will be set to the value of the **Hint (Tooltip)** property of the item, if the property is a literal string value. If JHeadstart Preferences Setting **Bind item property to corresponding ADF BC Control Hint?** is checked, the **Hint (Tooltip)** item property will be set to the EL expression that references this control hint: `#{SHINTS$.tooltip}`.

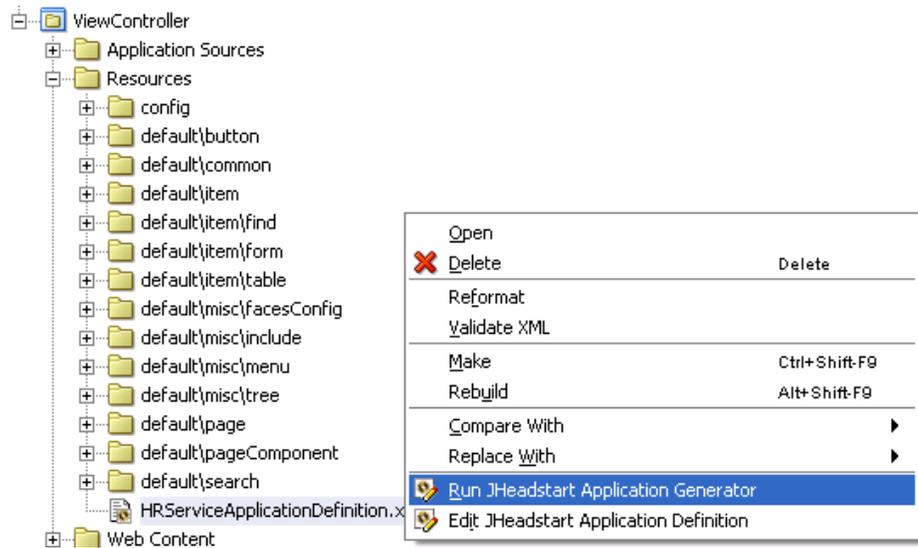
4.5. Running the JHeadstart Application Generator

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”.

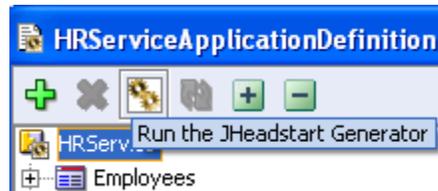
Once you have got the service and group definitions right, you can generate the application.

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

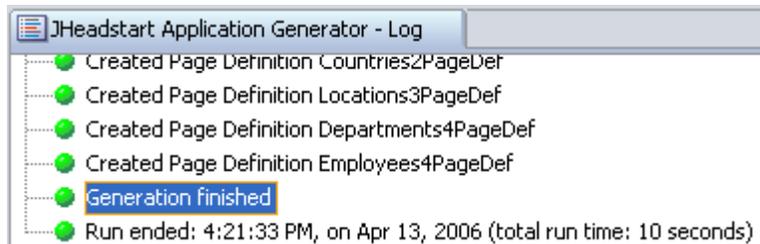
1. Right click the Application Definition File in the Applications Navigator, and choose Run JHeadstart Application Generator.



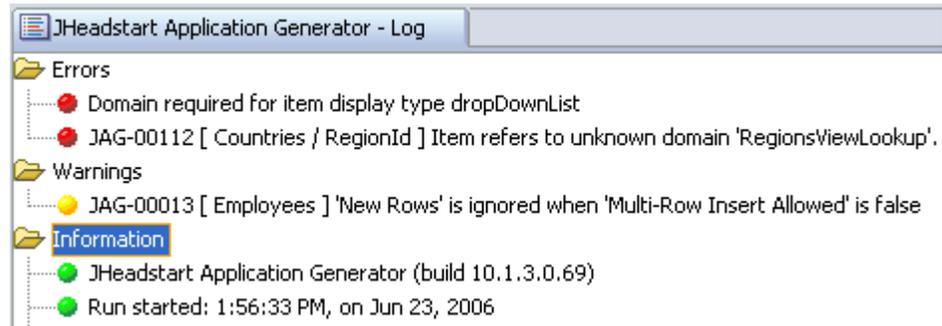
2. From within the JHeadstart Application Definition editor, click the third button from the left.



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**.

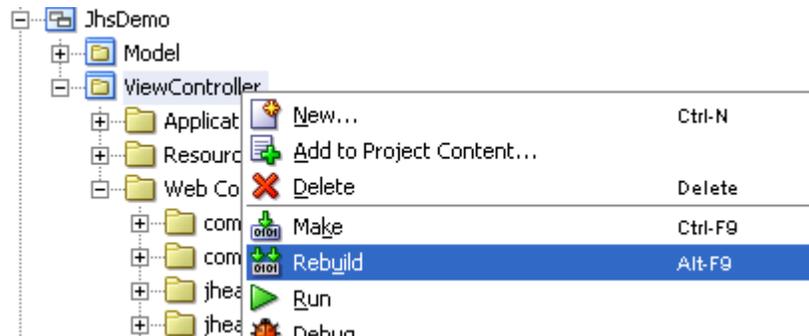


4.6. Running the Generated Application

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

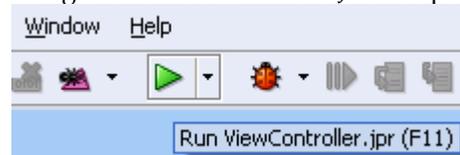


- Click the Rebuild icon in the JDeveloper toolbar



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),

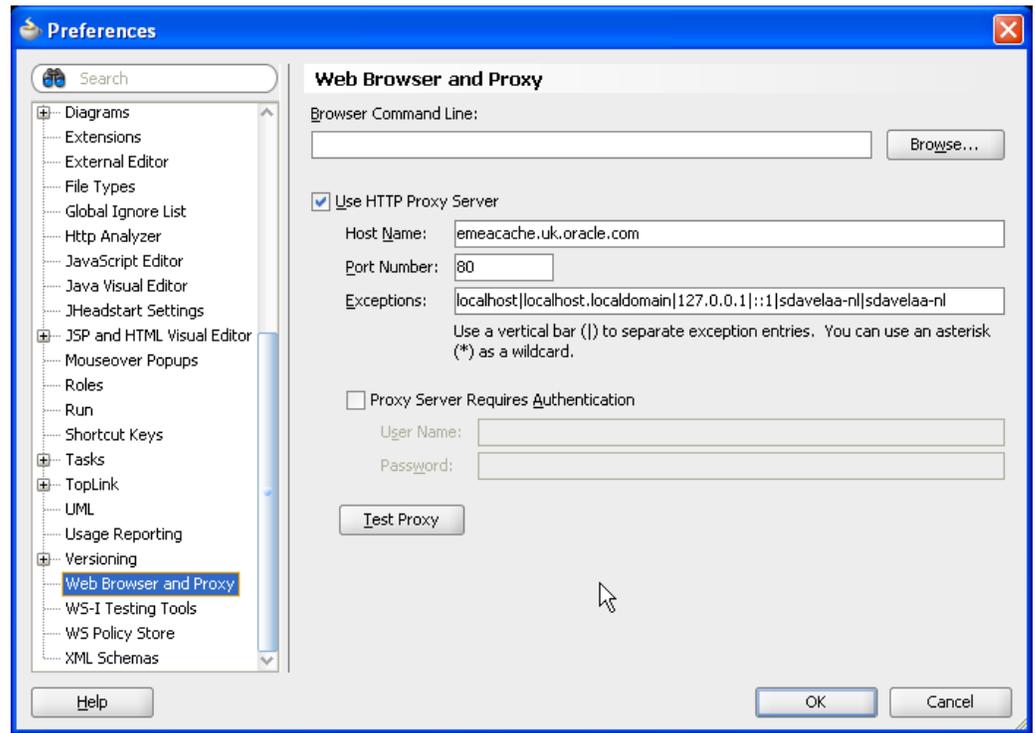


The JHeadstart Application Generator has automatically set the first generated page of the first group to be the default run target

- To directly run a specific page, select one of the generated .jsf files in the Application Navigator, right-mouse-click and choose Run.
- Open the 'faces-config.xml' in Diagram view and right click a .jsf 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 WebLogic Server. Go to the menu option **Tools -> Preferences**, and click on **Web Browser and Proxy** and check the proxy settings.



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 original 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 Page Fragment?**
- Group property **Generate Search Area in Page Fragment?**
- Item Group Region property **Generate in Page Fragment?**

Generation Settings	
Generate Pages? *	<input checked="" type="checkbox"/>
Generate Group in Page Fragment? *	<input type="checkbox"/>
Generate Search Area in Page Fragment? *	<input type="checkbox"/>
Generate Page Definition? *	<input checked="" type="checkbox"/>
Clear Page Definition Before Generation? *	<input checked="" type="checkbox"/>
Overwrite Page Definition Bindings? *	<input checked="" type="checkbox"/>
Generate Group Taskflow? *	<input checked="" type="checkbox"/>
Always Passivate State Before Commit? *	<input type="checkbox"/>

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 page fragments by checking the **Generate Group in Page Fragment?** 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 Page Fragment?**. 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 Page Fragment?**.

4.7. 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 (*.jsf)	UI Pages Directory property at service level	JSF files define the application page using ADF Faces.
ADF Faces Fragments (*.jsff)	UI Page Regions Directory property at service level	ADF Faces fragments 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.
adfc-config-[ServiceName].xml	Service ADFc Config property at service level	ADF Faces Navigation Rules between groups.
JhsCommon-beans.xml (or other name)	Common Beans Config property at application level	JSF managed beans that are common to every group in the Application Definition
Adfc-config-[GroupName].xml	Group ADFc Config Directory property at service level	ADF Faces managed beans specific to groups in the Application Definition, plus the navigation rules between the pages in a group.
Menu_root.xml Menu_[ServiceName].xml	(Root) Menu Model File property at application/service level	Defines the menu that is visible for the entire application and each service.

This page is intentionally left blank.



Generating Page Layouts

This chapter describes what the various view type and group usages are that JHeadstart can generate. It describes the following group usages:

- [Standalone Pages](#)
- [Reusable Regions with Page Fragments](#)
- [List of Values Window](#)

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

- [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 create more advanced page layouts by [Controlling Page Layout Using Region Containers, Item and Group Regions](#).

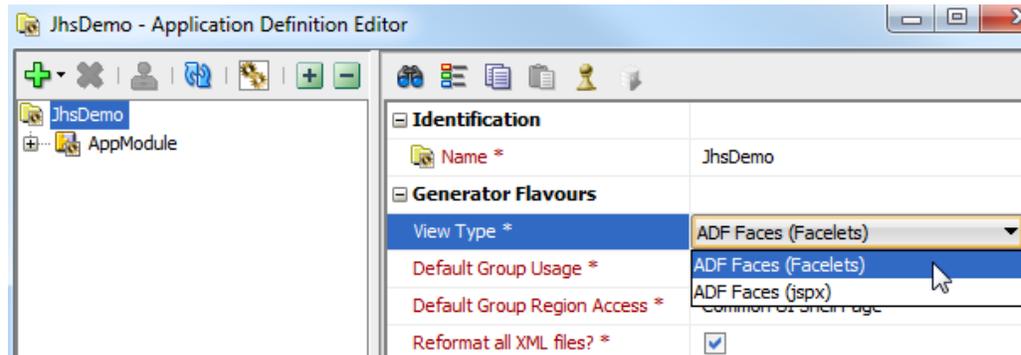
5.1. Choosing a View Type and Group Usage

JDeveloper 11.1.2 supports two ways of building JSF pages. You can choose to create either a Facelets page or a JSP page. Facelet pages use the extension *.jsf. Facelets is a JSF-centric declarative XML view definition technology that provides an alternative to using the JSP engine. JSP pages use the extension .jspx.

We recommend using Facelets to take advantage of the following:

- The Facelets layer was created specifically for JSF, which results in reduced overhead and improved performance during tag compilation and execution.
- Facelets is considered the primary view definition technology in JSF 2.0.
- Some future performance enhancements to the JSF standard will only be available with Facelets.

JHeadstart supports both page flavors through the **View Type** property that can be set at the application level in the JHeadstart Application Definition editor.



Following the above recommendation, we suggest you leave the **View Type** property to its default of “ADF Faces (Facelets)”.

For each top level group, a “group usage” can be selected. As the name suggests, it tells JHeadstart how a certain group will be *used* within your application. Especially the new ‘reusable region’ usage is a new and interesting feature introduced with JHeadstart 11, allowing for more reuse within your application.

JHeadstart offers 3 different ways in which your group can be used within the application:

- Stand-alone Pages
- Reusable Region with Page Fragments (Recommended)
- List of Values Window

The following paragraphs describe what the best option is in your situation and what happens when you generate a group with a particular group usage.



Suggestion: There is a service level property called **Default Group Usage**. If you specify a value there, you will not need to do this at the group level.

5.1.1. Stand-alone Pages

When you generate a group as stand-alone pages, each page will be generated as a normal JSF file (not a fragment). It is therefore not reusable inside other pages.

A stand-alone page is probably a bit easier to understand than a region with page fragments, but is also less flexible.

There are two main reasons why stand-alone pages are less flexible compared to regions with page fragments:

1. The generated pages are not reusable. Because the task flow is bounded but does not use fragments, it cannot be used as a region inside other pages.
2. JHeadstart 11 generates a bounded task flow for each top-level group. For standalone pages, this means that the **entire** page is within that bounded task flow (including the top menu, global buttons, etc). Global navigation rules are therefore not directly accessible within such a page. We have created a generic solution for this problem, though.



Reference: This generic solution to abandon implicitly bounded task flows is documented on the JHeadstart blog:

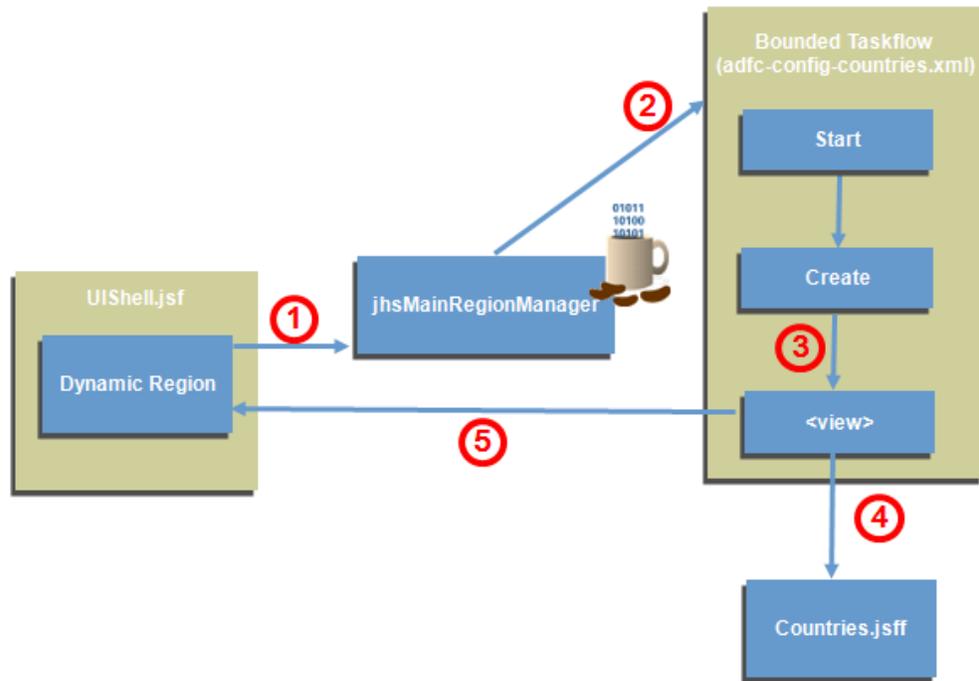
http://blogs.oracle.com/jheadstart/2009/04/core_adf_11_abandoning_a_bound.html

5.1.2. Region with Page Fragments (Recommended)

When you choose this option, the pages will not be generated as stand-alone pages (.jsf) but as page fragments (.jsff). The fragment only contains the basic 'content' of the group (i.e. the form or table itself) and not the common items on the screen (such as the menu).

A UIShell page will also be generated, that will act as the container for the fragment. Therefore the UIShell page only contains the common items on the screen (such as the menu) and not the content.

To understand how it all works together, we take a look at the following diagram, which represents the main elements involved in showing a region with page fragments at runtime. To make it less abstract, it shows the actual files if you would generate a (form based) group called 'Countries'.



The end result for the user is that (s)he will see the page UIShell.jsf with the content of Dynamic Region set to Countries.jsff.

But how does it work?

Step 1

The UIShell page does not contain the fragment (Countries.jsff) directly, but defines the following region:

```
<f:facet name="pageContent">
  <af:region id="mainRegion" value="#{bindings.mainRegion.regionModel}"
    partialTriggers="::Menu1 ::Menu2 ::Dmenu1 ::Dmenu2 ::pendingChangesDialog" />
</f:facet>
```

So, the region gets its value from the bindings layer, in other words, the page definition file (UIShellPageDef.xml). If we look there, we see the following binding defined:

```
<taskFlow id="mainRegion"
  taskFlowId="{pageFlowScope.jhsMainRegionManager.currentTaskFlowId}"
  parametersMap="{pageFlowScope.jhsMainRegionManager.currentParamMap}"
  RefreshCondition="{pageFlowScope.jhsMainRegionManager.currentParamMapChanged}"
  xmlns="http://xmlns.oracle.com/adf/controller/binding"/>
```

Here we see the content of the region is determined by a taskflow.



Reference: An important concept in ADF 11 is “task flows”. The Fusion Developers Guide has 6 chapters about it, bundled in Part IV:

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/partpage4.htm#BHAGEFJF

The region is also a *dynamic* region, because the taskFlowId is not hardcoded, but derived from a managed bean ‘jhsMainRegionManager’. The idea behind this bean is that it

points to the current task flow that should be visible within the UIShell page. So neither the page, nor the binding file gives us any clue on what the content for the region will be. This is all stored in `jhsMainRegionManager`, which also stores the parameters for the task flow.

Step 2

On each request, ADF will re-evaluate the value of the attribute `'currentTaskFlowId'` to see if it needs to 'reload' the content of the region. Also when the attribute `'currentParamMapChanged'` is set to `'true'` (same taskflow, but different parameters) the region will be refreshed.

The taskflow-id looks like:

```
/WEB-INF/adfc-config-Countries.xml#CountriesTaskFlow
```

It contains the whole path to the ADFC config file that contains the task flow, plus the ID of the task flow that needs to be started.

Now, if you click a menu tab on the top of the page, it will set the value of `'currentTaskFlowId'` to another value, which will cause ADF to start another task flow and reload the region.

Step 3

In this step we enter the task flow itself, defined in the adfc-config XML file. A JHeadstart generated bounded task flow always contains a start activity (which is the default activity), plus a create activity and a `SetCurrentRow` activity. The create activity will be executed in case of a wizard, for example. Then you want to create a row first, before displaying the page. `SetCurrentRow` can be used for deep linking, i.e. going directly to a specific row in the view object, not the first row.

After these activities are complete, the `'firstPage'` activity will be executed. This is wired to the initial `<view>` that your group will display (the Table page, if you have Table-Form layout for example). In this case it is simply the form page called Countries:

```
<control-flow-case>
  <from-outcome>firstPage</from-outcome>
  <to-activity-id>Countries</to-activity-id>
</control-flow-case>
```

Step 4

At this point we know which fragment needs to be loaded for our region.

```
<view id="Countries">
  <page>/pages/Countries.jsff</page>
</view>
```

ADF will now fetch the contents of the JSFF file and the task flow will now 'pause' here, until the user will do another action.

Notice that the page fragment (`Countries.jsff`) has another page definition file than the UIShell. Therefore the `Countries` task flow is really independent of the context in which it is used.

Step 5

The content of the fragment that was found, is put inside of the <af:region> and displayed to the user.

5.1.2.1. Common vs. Group UIShell page

When the group usage is set to 'Region with Page Fragments' JHeadstart gives another choice:

- Generate one UIShell page **for the whole application**. This means that this UIShell page will be the single JSF file used; all other pages in your application are fragments (JSFF) that are shown within this page. This approach generates the fewest number of files and is the fastest at runtime.
- Generate a UIShell page **per group**. The filename of this page will simply be *GroupName.jsf* (e.g. *Countries.jsf*). This group page will work in the same way as the application-wide UIShell page; except that if you navigate to another group, it will not simply refresh its region but really navigate to that page. This approach generates one file extra per group (the group UIShell page). It is also slightly slower at runtime, because navigating between groups means that not only the region needs to be refreshed, but the entire page. Use the option if the page content surrounding the group content should be different from the standard UIShell content.

5.1.3. Reusing Groups

JHeadstart does not only generate regions with fragments, it also supports reusing them inside other pages.

Suppose we have an Employees group that will display all Employees in a table layout, which can be accessed via the top level menu. We also have a Departments group that needs to display the employees for the current department as well. So we need to display employees at two levels:

1. We want to see and edit *all* employees via the top level menu entry
2. We want to see and edit the Employees *for a certain department* on the Departments page.

In JHeadstart, we now have two options to do this:

1. Create two groups for the Employees: one top level group, and one nested detail group inside Departments.
2. Create one top level group for Employees, and *reuse* that group within the Departments group.

The first option is perhaps the most straightforward; but it also contains a complete (and redundant) copy of the Employees group. Especially when you need to do a lot of maintenance or heavy customizations on these groups this is not desirable.

The second option is much more 'clean'. It allows for a complete reuse of the Employees group. Of course, we need to do little adjustments to the group to make it fully reusable;

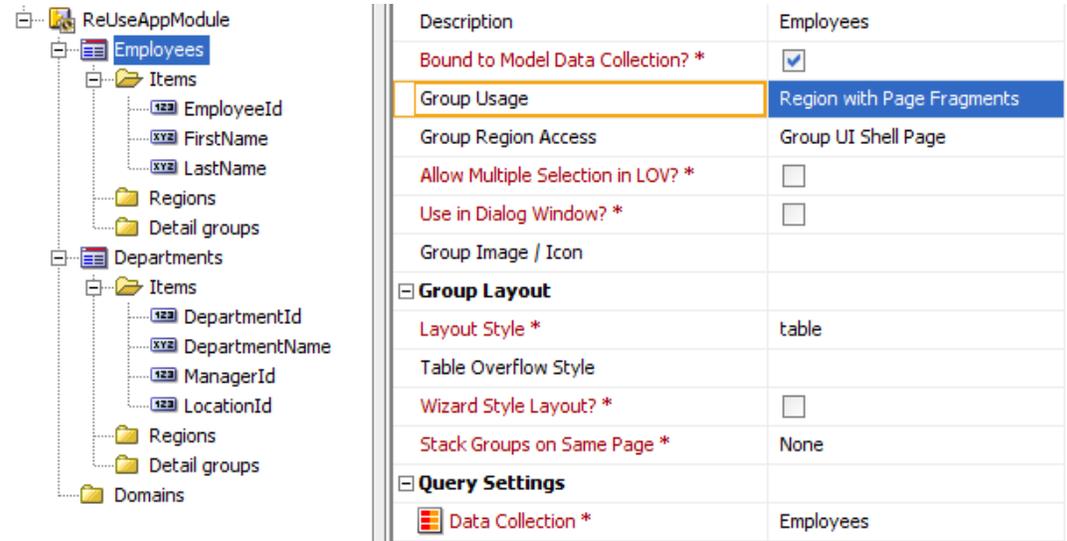
for example, the group needs to be able to show only the employees for a certain department.

To re-use a group inside another group, follow the following steps:

5.1.3.1. Create the reusable group

First create the group that you want to re-use, with group-usage set to “Region with Page Fragments”. In the example below we want to re-use the Employees group in our application, so make sure the group usage is set correctly.

The group that *includes* the re-usable group (such as Departments) does **not** have to be a reusable group.

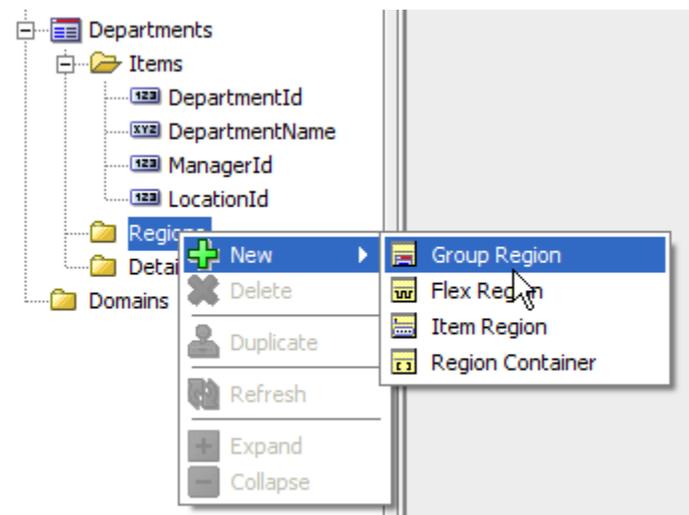


The screenshot shows the application tree on the left and the properties window on the right. The application tree shows a hierarchy starting with 'ReUseAppModule', containing 'Employees' and 'Departments' groups. The 'Employees' group has sub-items: 'Items' (with 'EmployeeId', 'FirstName', 'LastName'), 'Regions', 'Detail groups', and 'Domains'. The 'Departments' group has sub-items: 'Items' (with 'DepartmentId', 'DepartmentName', 'ManagerId', 'LocationId'), 'Regions', 'Detail groups', and 'Domains'. The properties window on the right shows the configuration for the 'Employees' group. The 'Group Usage' property is set to 'Region with Page Fragments'. Other properties include 'Bound to Model Data Collection?' (checked), 'Group Region Access' (Group UI Shell Page), 'Allow Multiple Selection in LOV?' (unchecked), 'Use in Dialog Window?' (unchecked), 'Group Image / Icon', 'Group Layout' (Layout Style: table, Wizard Style Layout? unchecked, Stack Groups on Same Page: None), and 'Query Settings' (Data Collection: Employees).

Description	Employees
Bound to Model Data Collection? *	<input checked="" type="checkbox"/>
Group Usage	Region with Page Fragments
Group Region Access	Group UI Shell Page
Allow Multiple Selection in LOV? *	<input type="checkbox"/>
Use in Dialog Window? *	<input type="checkbox"/>
Group Image / Icon	
Group Layout	
Layout Style *	table
Table Overflow Style	
Wizard Style Layout? *	<input type="checkbox"/>
Stack Groups on Same Page *	None
Query Settings	
Data Collection *	Employees

5.1.3.2. Add reusable group inside other group

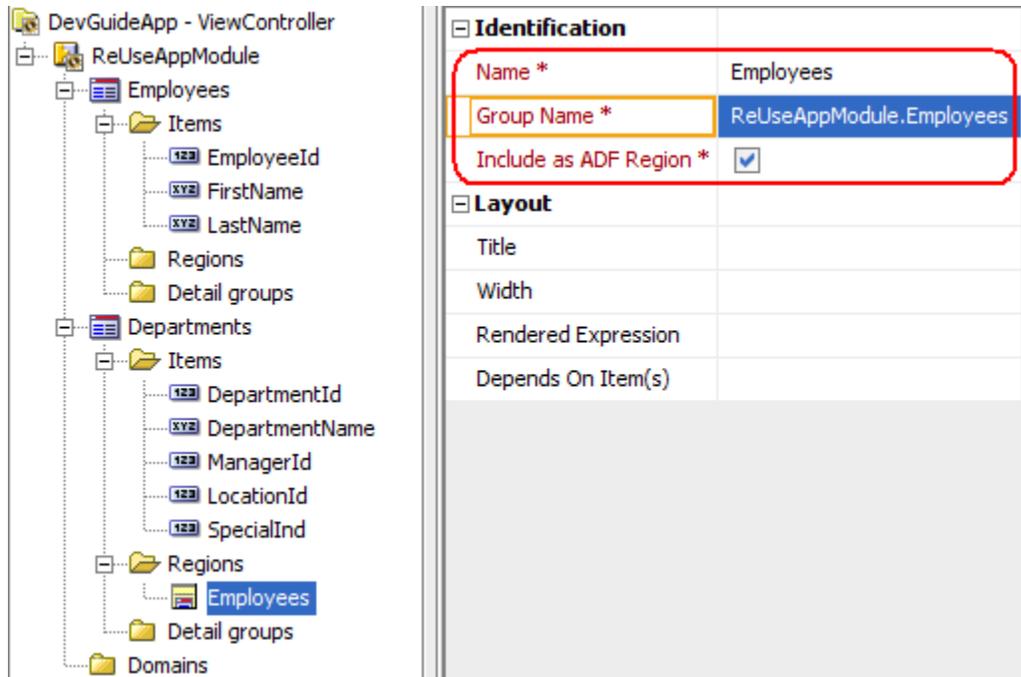
Locate or create the group that will contain your reusable group (the Departments group in this example). Right click the Regions folder, and create a Group Region.



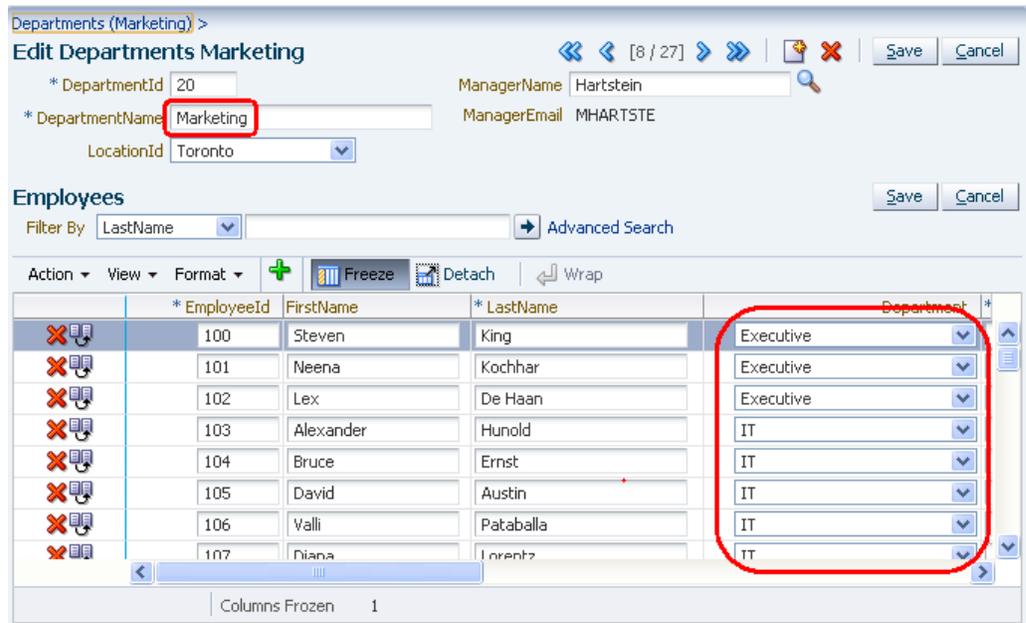
The screenshot shows the application tree with the 'Regions' folder selected under the 'Departments' group. A context menu is open over the 'Regions' folder, showing options: 'New', 'Delete', 'Duplicate', 'Refresh', 'Expand', and 'Collapse'. The 'New' option is expanded, showing sub-options: 'Group Region', 'Flex Region', 'Item Region', and 'Region Container'. A mouse cursor is pointing at the 'Group Region' option.

Give the new Group Region a name, and mark the checkbox ‘Include as ADF Region’. Now you can select your reusable group from the dropdown box at ‘Group Name’.

Notice that the reusable group does not have to reside in the same Service; you can choose any reusable group within your application!



We can generate our application now, which will give us the following Departments page:



So we have already successfully included an Employees table in the Departments page.

However, it needs some customizations. We see **all** the employees, plus we see another Save/Cancel button and a search region. This is not what we want in this case.

However, removing the buttons from the Employees region would also remove them from the top-level page. We do not want this to happen. Therefore we need to make the

Employees region more flexible so we can dynamically set the 'scope' of employees (all or a specific department) and also show or hide the buttons and search area dynamically.

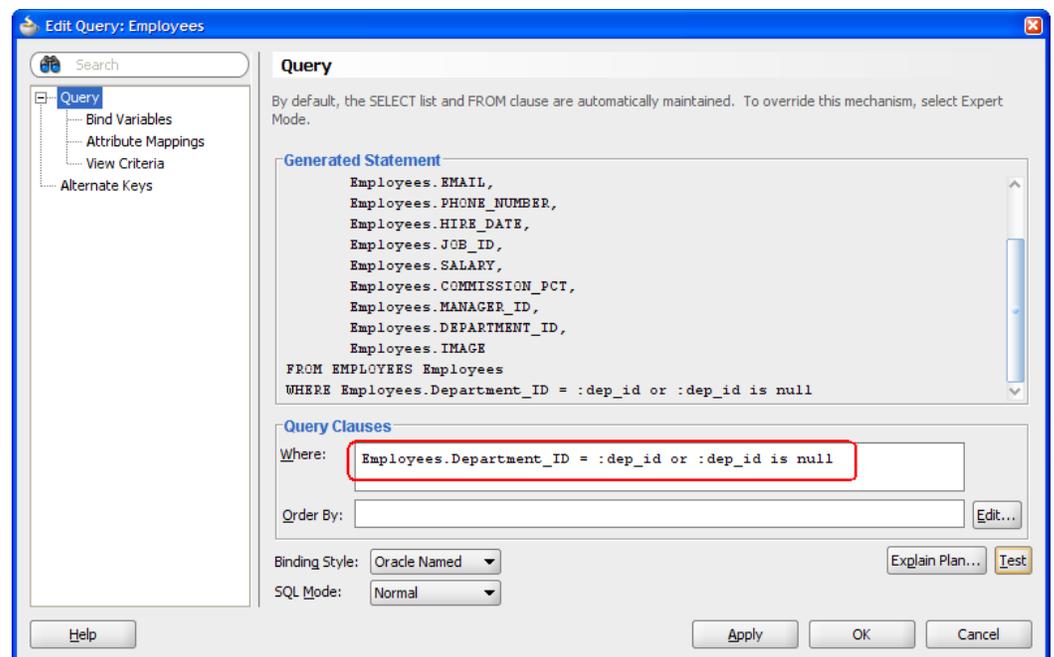
To restrict the query when used as a child of the Departments group, we have two options:

- We can set up an optional department id bind parameter on the view object and pass that in when used as child of the Departments group.
- We can use the **Data Collection Expression** group property to dynamically switch the view object usage of the employees iterator binding at runtime.

Both techniques are explained below. While more advanced, the second technique is generally easier and faster to implement.

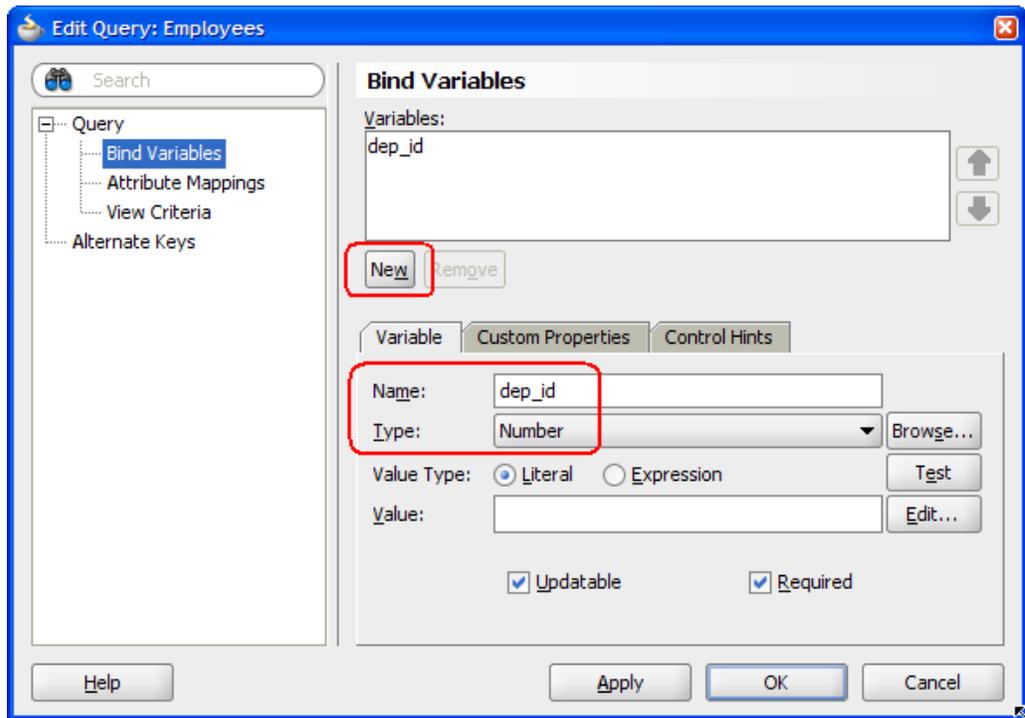
5.1.3.3. Using a Bind Parameter to Restrict the Employees Query

Double click your View Object and go to the Query tab. Press the Edit icon and type in the new Where Clause:



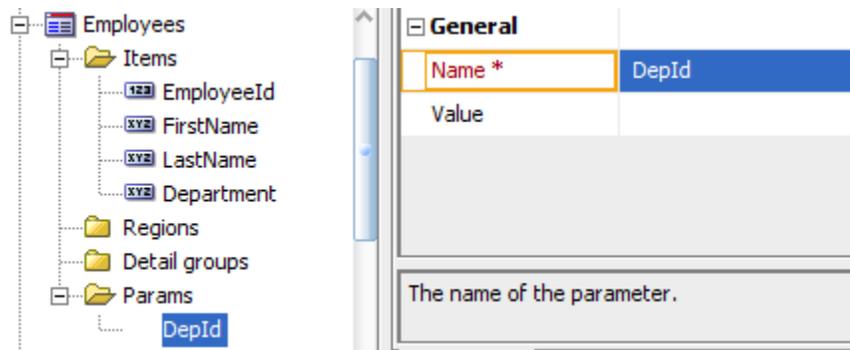
We introduced a bind parameter `dep_id` that can either be NULL (then every employee will be returned) or contain a `Department_ID`. In the latter case only the employees for the department with that ID will be returned.

We also need to define our `dep_id` as a bind variable, by clicking **Bind Variables** in the left column. Click **New** and set the name and type of the variable to the correct values:



Now our view object is ready to handle both situations: standalone and as a sub-view of departments.

Now that we have defined our bind variable, we need to set its value. Go to the reusable group, and create a Parameter:



When JHeadstart generates this group, it will now introduce a new task flow input parameter called 'DepId', which can be accessed with the EL statement `{pageFlowScope.DepId}`.

This is the value we want to put in our bind param, so we go to the `Employees` group and set the bind parameter to this value. The dropdown box knows about our defined input parameter, and already gives you the value. All you need to do is prefix this value with `dep_id=`, because we are using named bind variables.

Query Settings	
Data Collection *	Employees
Data Collection Implementatio...	Employees
Query Bind Parameters	dep_id=#{pageFlowScope.DepId}
Requery Condition	
Search Settings	#{pageFlowScope.DepId}

We are finished with our customizations on the `Employees` group. It is flexible enough to show all rows (when the parameter `DepId` is empty) or to show only the employees for a specific department (when `DepId` is not null).



Suggestion: The top level group `Employees` will show all `Employees` in this case, because the parameter '`DepId`' is by default empty (null). If you specify a value for the parameter however, it will be used as the *default* value. So if you enter a value like 20 (Marketing Department) you will only see the `Employees` of the Marketing Department on the top level `Employees` page.

To set the value of the `DepId` parameter on the `Departments` page, we go to the `EmployeesInDepartment` group region inside the `Departments` group. Use the green + icon or right-click the group region to add a parameter.

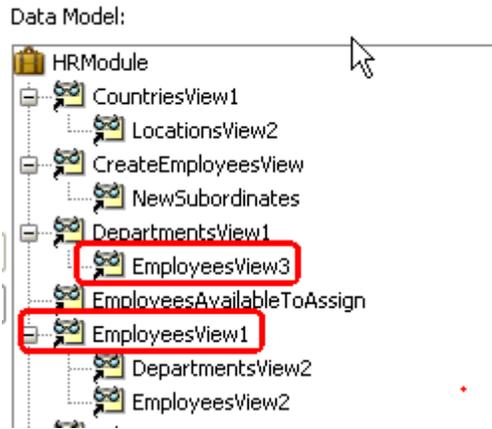
All available taskflow parameters are shown in the combo box when you click on the parameter name, so you can select "DepId" from the list. We want to set the value of the parameter to the current `DepartmentId`, which is `#{bindings.DepartmentsDepartmentId.inputValue}`. This is same expression as you will find in the 'value' attribute of the `DepartmentId` component inside the JSF page.

The screenshot shows the application editor interface. On the left, a tree view displays the 'Departments' group containing 'Items', 'Regions', 'EmployeesInDepartment', 'Params', and 'Detail groups'. The 'DepId' parameter is highlighted under 'Params'. On the right, the 'General' property window is open, showing the following properties:

General	
Name *	DepId
Value	#{bindings.DepartmentsDepartmentId.inputValue}

5.1.3.4. Using the Data Collection Expression Property

When you initially create an application based on the HR application, you get a separate `Employees3` group as a detail group of the `Departments` group. This `Employees3` group is based on the `EmployeesView3` view object usage, which is a nested usage in the application module, as shown below.

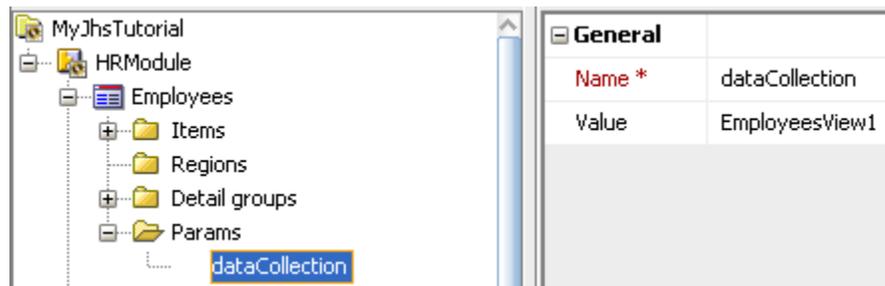


Because EmployeesView3 is nested, ADF Business Components takes care of the master-detail synchronization as explained in [section 5.6 Creating Master-Detail Pages](#).

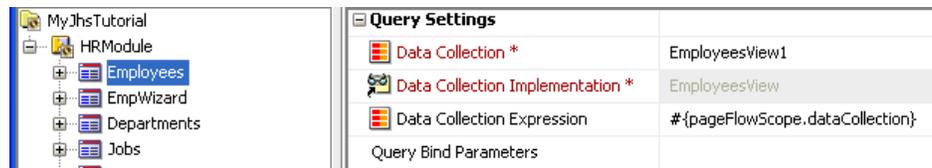
By using the group-level **Data Collection Expression** property we can switch at runtime the view object usage of the employees iterator binding: when the employees group is used as a top-level group, it should use EmployeesView1 view object usage, when used as a detail of Departments group, it should use the EmployeesView3 view object usage.

Here are the steps to do this:

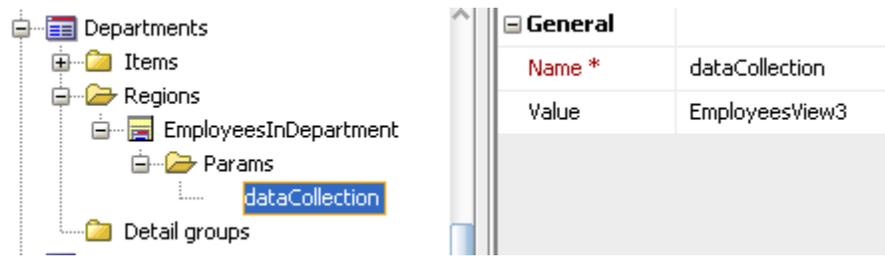
- We first define a parameter named dataCollection for the Employees group, with the (default) value set to EmployeesView1



- Still in the Employees group, we set the **Data Collection Expression** to the value of the dataCollection taskflow parameter which can be picked from the combo box.



- In the Departments group, we go to the EmployeesInDepartment group region, and we add the same dataCollection parameter, and set the value to EmployeesView3.



That's all! If you are curious how this works at runtime, then take a look at the EmployeesIterator generated into the EmployeesPageDef:

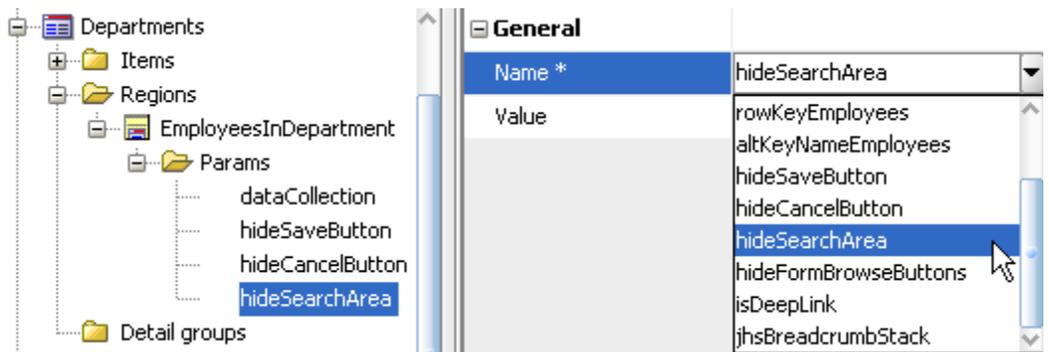
```
<iterator id="EmployeesIterator" Binds="#{pageFlowScope.dataCollection}"
DataControl="HRModuleDataControl" RangeSize="10"/>
```

As you can see, the **Binds** property of the iterator binding does not contain the hard coded value of an employee view object usage, but now gets the value from our task flow parameter.

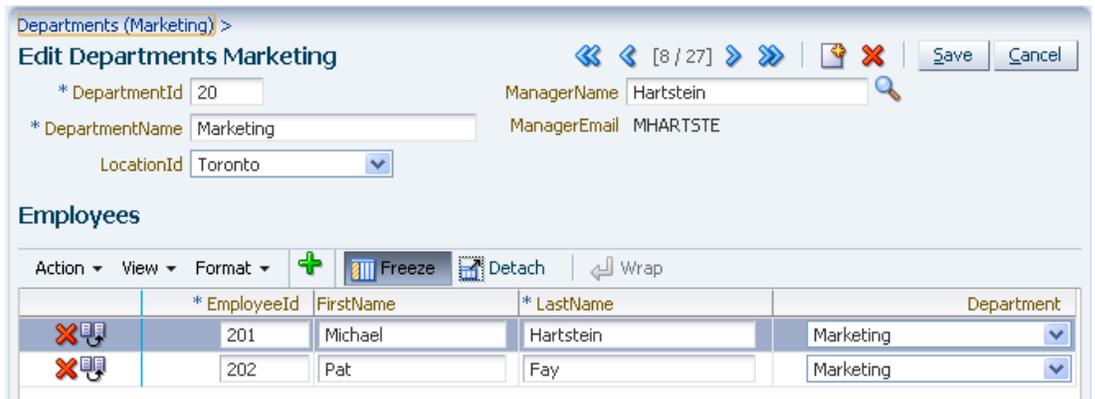
5.1.3.5. Hiding Save/Cancel Buttons and the Search Area

On the Departments page, there were also unnecessary Save/Cancel buttons for the Employees table, as well as a search area that we don't need in this case.

Fortunately, every reusable group has predefined task flow parameters for hiding those elements. When you add a new parameter, and click on the name property, the combo box list already shows all available task flow parameters. Add the parameters hideSaveButton, hideCancelButton and hideSearchArea and set the value to true.



When we regenerate our JHeadstart application, we get the desired result:



The Save/Cancel buttons are gone, the search area is no longer visible and only those employees are shown that belong to the current Department.

You might argue that when used as a child of the departments group, we do not want to see the `DepartmentId` column at all in the employees table. Well, this is easy to do, either by adding an additional taskflow parameter, or using an existing one. For example, when using the `dataCollection` parameter to restrict the employees query, we know that the `DepartmentId` column should be hidden when the value of the `dataCollection` parameter is "EmployeesView3". This can be achieved by setting the **Display in Table Layout** property of `DepartmentId` item to `# {pageFlowScope.dataCollection != 'EmployeesView3' }`

5.1.4. List of Values Window

When you choose List of Values, JHeadstart will generate:

- One adfc config file with a bounded task flow, plus the beans for the List of Values component
- One page fragment
- No UI Shell page

The LOV is therefore a completely reusable component throughout your application. For more information about the JHeadstart LOV, see Chapter 6, section "Generating a List of Values (LOV)".

5.1.5. Showing a Group in a Popup Window

It is also possible to show the content of a group in a dialog popup window. This can be done in two ways:

- By defining an item in the base group with **Display Type** "groupLink". See chapter 6, section "Navigating Context-Sensitive to Another Group Taskflow (Deep Linking)" for more information.
- By defining a group region inside a region container with display style "Modal Popup Window" or "Modeless Popup Window" See section 5.10.4 [Generating Content in a Popup Window](#)

5.2. 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.

To generate a Form Page you 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	
Label Alignment	At left of item
Form Width	
Label Width	
Field Width	
Columns *	2
Rows *	1

- The **Label Alignment** property can be used to position the labels at the left or above the items.
- The **Forms Width** property determines the amount of horizontal space the form layout can consume on the page as a percentage or in pixels. 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.
- The **Label Width** property defines the preferred width of the labels. The web browser may override this dimension if it cannot fit the labels in the space allocated. You may define the **Label Width** using any CSS unit such as em, px, or %. The units used for this value must be identical to the units used in the **Field Width** value. If a **Label Width** is provided as a percentage **Label Width** and the **Field Width** should total up to 100% (regardless of the number of columns). If the **Label Width** is not specified, the browser will let the children components have a natural flowing layout. However, if the **Label Width** is not specified but a **Field Width** is specified as a percentage, the **Label Width** will be derived as the appropriate percentage value for you.
- The **Field Width** property defines the preferred width of the fields. The web browser may override this dimension if it cannot fit the fields in the space allocated. You may define the **Field Width** using any CSS unit such as em, px, or %. The units used for this value must be identical to the units used in the **Label Width** value. If a **Field Width** is provided as a percentage the **Label Width** and the **Field Width** should total up to 100% (regardless of the number of columns). If the **Field Width** is not specified, the browser will let the children components have a natural flowing layout. However, if the **Field Width** is not specified but a **Label Width** is specified as a percentage, the **Field Width** will be derived as the appropriate percentage value for you.
- The **Columns** property defines the maximum number of columns to show.

- The **Rows** property defines the number of rows after which to start a new column. The number of rows actually rendered depends also on the **Columns** property. If the children will not fit in the given number of rows and columns, the number of rows will increase to accommodate the children.

Normally, you can leave the **Rows** property to the default of 1, and only specify the **Columns** property. ADF faces will then equally distribute the items over the number of columns specified, as shown below.

The screenshot shows a form titled "Edit Employee Pataballa" with the following fields arranged in a 2x2 grid:

- Top-left: * EmployeeId (106), FirstName (Valli), * LastName (Pataballa), Department (IT), * Email (VPATABAL), PhoneNumber (590.423.4560)
- Top-right: * HireDate (05-Feb-1998), * JobId (IT_PROG), * Salary (4800), CommissionPct
- Bottom-left: (empty)
- Bottom-right: ManagerId (Hunold)

If you want the first column to have more items, then you should set the **Rows** property. For example, the layout shown below is generated with **Rows** set to 7 and **Columns** to 2.

The screenshot shows the same form titled "Edit Employee Pataballa" but with a different layout. The fields are arranged in two columns:

- Column 1 (left): * EmployeeId (106), FirstName (Valli), * LastName (Pataballa), Department (IT), * Email (VPATABAL), PhoneNumber (590.423.4560)
- Column 2 (right): * JobId (IT_PROG), * Salary (4800), CommissionPct, ManagerId (Hunold), * HireDate (05-Feb-1998)

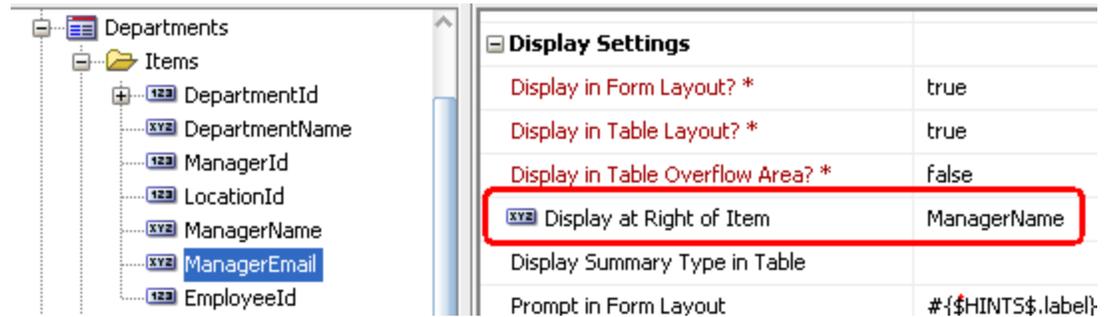
See section [Controlling Page Layout Using Region Containers, Item and Group Regions](#) for more info on how you can group items on the page and have more control over the relative positioning of these item groups.

 **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 laid out from top to bottom, and left to right.

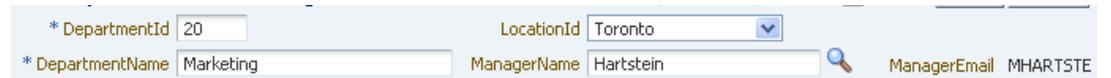
 **Attention:** The order of the rows when navigating to the next or previous record is determined by the Order By clause in the View Object. See section 3.3.5 - Determining the Order of Displayed Rows.

5.2.1. Displaying an Item at the Right of Another Item

Sometimes you want to position an item directly at the right of another item, regardless of the number of columns specified for the form layout. For example, zip code and city should always be displayed beside each other, or when using LOV items, additional items copied back by the LOV should be displayed at the right of the LOV item. To achieve this you can use the **Display at Right of Item** property.

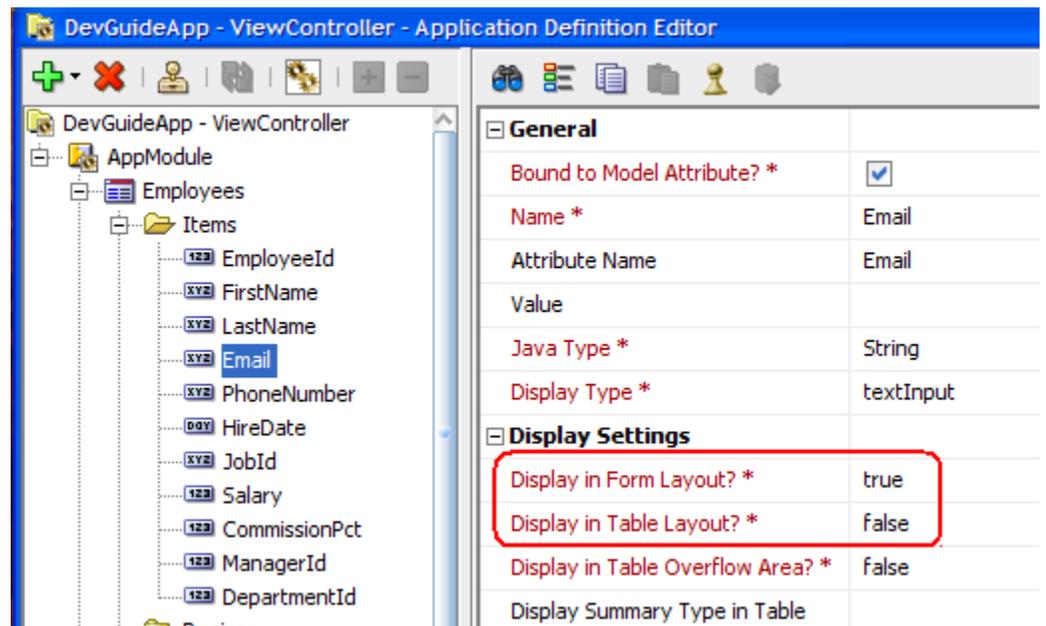


In the above screen shot, the `ManagerEmail` item is configured to be displayed at the right of the `ManagerName` item, which results in a generated layout as shown below.



5.2.2. Hiding 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. A custom EL expression that is evaluated at runtime (must return true or false). For example, by using an EL expression, you can conditionally show/hide the item based on the value of another item, or based on task flow input parameters.

5.2.3. 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 form 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.
4. Quick/Advanced search area is hidden.

Page in 'Update' mode:

The screenshot shows a form titled "Edit Employee Pataballa". The form contains several input fields and dropdown menus. The fields are: EmployeeId (106), HireDate (05-Feb-1998), JobId (IT_PROG), Salary (4800), ManagerId (Hunold), CommissionPct (empty), Department (IT), Email (VPATABAL), FirstName (Valli), LastName (Pataballa), and PhoneNumber (590.423.4560). There are navigation buttons (back, forward, search) and "Save" and "Cancel" buttons.

The same page in 'Create' mode:

The screenshot shows a form titled "Enter New Employee". The form contains several empty input fields and dropdown menus. The fields are: EmployeeId, HireDate, JobId, Salary, ManagerId, CommissionPct, Department, Email, FirstName, LastName, and PhoneNumber. There are "Save" and "Cancel" buttons.

When pressing a 'New' button, the `execute` method on the `CreateRowBean` is executed which in turn invokes the 'CreateInsert' action in the paged definition. The `execute` method stores an entry in a 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:

```
#{pageFlowScope.createModes.CreateEmployees}
```

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:panelHeader text="#{pageFlowScope.createModes.CreateEmployees ?
nls['INSERT_TITLE_EMPLOYEESFORM'] :
nls['EDIT_TITLE_EMPLOYEESFOR:#{bindings.EmployeesLastName}'] }"
```

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

```
rendered="#{!pageFlowScope.createModes.CreateEmployees}"
```

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

5.2.4. Controlling the Page Title in Form Layout

JHeadstart uses the **Display Title Singular** property to determine the title in form layouts. If this property is not set, it uses the value of the **Display Title Plural** property.

You will have noticed that JHeadstart also adds a verb in front of the title based on whether the form is in create, update or view mode. If you do not want this verb to be added, and you just want to use the value of the **Display Title Singular** property, then you should uncheck the checkbox property **Add Verb to Form Title?**.

Labels	
Tabname	Employees
Display Title (Plural) *	Employees
Display Title (Singular)	Employee
Show Display Title? *	<input checked="" type="checkbox"/>
Add Verb to Form Title? *	<input type="checkbox"/> *
 Descriptor Item *	LastName

If you do not want to display a title at all, you can uncheck checkbox property **Show Display Title?**.

5.3. 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 value should be displayed in the list box 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.



Attention: The order of the rows in the list box 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.

5.4. Creating Table Pages

In many situations you want to present multiple records to the user in one page. You can generate this type of page in a number of ways:

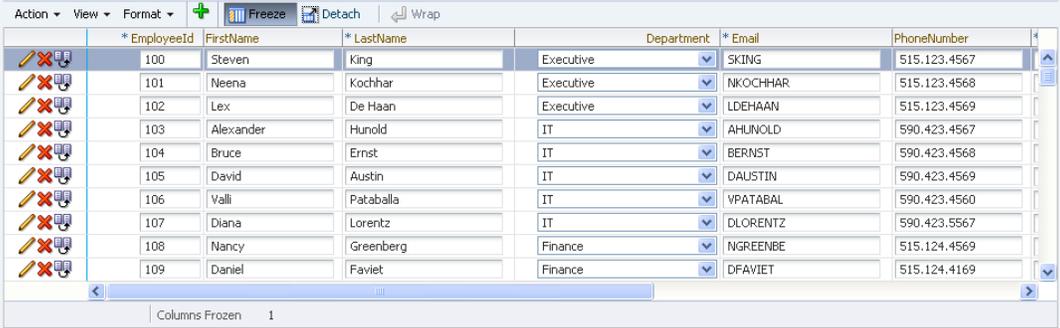
1. Using a Table Page. In a Table Page the data can be manipulated directly in the table. Use this option when the number of items in the group is small, so the table fits on the page. See the remainder of this section.
2. Using a Table Page with table overflow. Use this when the number of items is too 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 [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 [Creating Table-Form Pages](#).

5.4.1. Using Table Layout and Stretching

Set the **Layout Style** property of your group to 'table' to generate a Table Page:

Group Layout	
Layout Style *	table
Table Overflow Style	
Wizard Style Layout? *	<input type="checkbox"/>
Stack Groups on Same Page *	None

By default the generated table looks as follows:



The screenshot shows a table with the following columns: EmployeeId, FirstName, LastName, Department, Email, and PhoneNumber. The table contains 10 rows of data, with the first row highlighted. The table is displayed in a window with a menu bar (Action, View, Format) and a toolbar (Freeze, Detach, Wrap). The status bar at the bottom indicates 'Columns Frozen 1'.

* EmployeeId	FirstName	* LastName	Department	* Email	PhoneNumber
100	Steven	King	Executive	SKING	515.123.4567
101	Neena	Kochhar	Executive	NKOCHHAR	515.123.4568
102	Lex	De Haan	Executive	LDEHAAN	515.123.4569
103	Alexander	Hunold	IT	AHUNOLD	590.423.4567
104	Bruce	Ernst	IT	BERNST	590.423.4568
105	David	Austin	IT	DAUSTIN	590.423.4569
106	Valli	Pataballa	IT	VPATABAL	590.423.4560
107	Diana	Lorentz	IT	DLORENTZ	590.423.5567
108	Nancy	Greenberg	Finance	NGREENBE	515.124.4569
109	Daniel	Faviet	Finance	DFAVIET	515.124.4169

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 number of rows displayed is based on the setting of the **Table Range Size** property which is set to 10 for the page shown above. The width of the table is determined by the **Table Width** property which by default is set to Compute. The **Table Width** property has three possible values:

- **Compute:** JHeadstart computes a default table width in pixels by looping over the items displayed in the table, and checking the item **Column Width** property, and if not set, the maximum length of the underlying view object attribute. The computed table width value has a maximum as specified by the application-level property **Maximum Table Width**.
- **Stretch:** Table stretches horizontally using the available space
- **A custom value in pixels.** This value can be typed in, and will be used as table width.

If you have a table with many columns, we recommend to set the **Table Width** to **Stretch**. When you do this, and the user resizes the browser window, the number of columns shown will change accordingly, and the horizontal table scrollbar will also adjust to allow you to see all the columns that are initially not visible. You can also stretch the table vertically; the number of rows displayed will then be dependent of the amount of vertical space available. Resizing the browser window will then also change the number of rows displayed in the table. To enable vertical stretching you check the group-level checkbox property **Enable Stretching?**. Note that whether the table stretching actually occurs depends on a number of rules that must be met, that are also documented in the help text of the **Enable Stretching?** Property. These rules can be summarized by saying that stretching is only possible if no other user interface items are displayed at the right or below the item within the same stretchable area.



Attention: Enabling horizontal and vertical stretching on tables is also the preferred solution to accommodate for the situation where various end users might use different screen resolutions.

If your table has so many columns that it is likely that a horizontal scrollbar will appear, you can configure which items should be frozen. Frozen items are always displayed at the left and do not disappear when the user starts using the horizontal scrollbar. The last frozen item can be set using the **Table Freeze Style** property, as shown below.



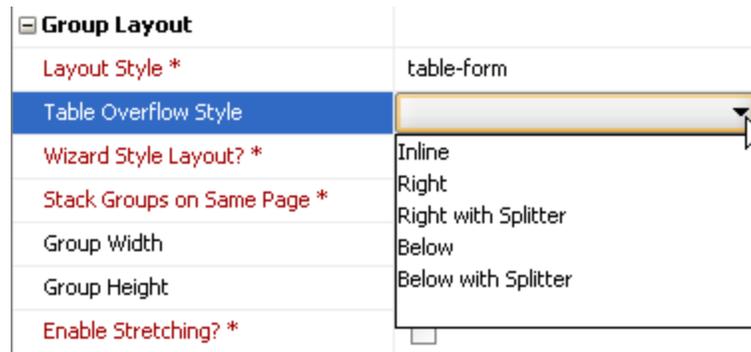
The screen shot below shows the table with the **Table Freeze Style** set to "Freeze up to Descriptor Item" and the **Descriptor Item** set to `LastName`.

* EmployeeId	FirstName	* LastName	* Email	PhoneNumber	* HireDate	* JobId
100	Steven	King	SKING	515.123.4567	17-Jun-1987	AD_PRES
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-Sep-1989	AD_VP
102	Lex	De Haan	LDEHAAN	515.123.4569	13-Jan-1993	AD_VP
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-Jan-1990	IT_PROG
104	Bruce	Ernst	BERNST	590.423.4568	21-May-1991	IT_PROG
105	David	Austin	DAUSTIN	590.423.4569	25-Jun-1997	IT_PROG
106	Valli	Pataballa	VPATABAL	590.423.4560	05-Feb-1998	IT_PROG
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-Feb-1999	IT_PROG
108	Nancy	Greenberg	NGREENBE	515.124.4569	17-Aug-1994	FI_MGR
109	Daniel	Faviet	DFAVIET	515.124.4169	16-Aug-1994	FI_ACCOUNT
110	John	Chen	JCHEN	515.124.4269	28-Sep-1997	FI_ACCOUNT
111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-Sep-1997	FI_ACCOUNT
112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-Mar-1998	FI_ACCOUNT
113	Luis	Popp	LPOPP	515.124.4567	07-Dec-1999	FI_ACCOUNT
114	Den	Raphaely	DRAPHEAL	515.127.4561	07-Dec-1994	PU_MAN
115	Alexander	Khoo	AKHOO	515.127.4562	18-May-1995	PU_CLERK
116	Shell	Baida	SBAIDA	515.127.4563	24-Dec-1997	PU_CLERK
117	Sigal	Tobias	STOBIAS	515.127.4564	24-Jul-1997	PU_CLERK

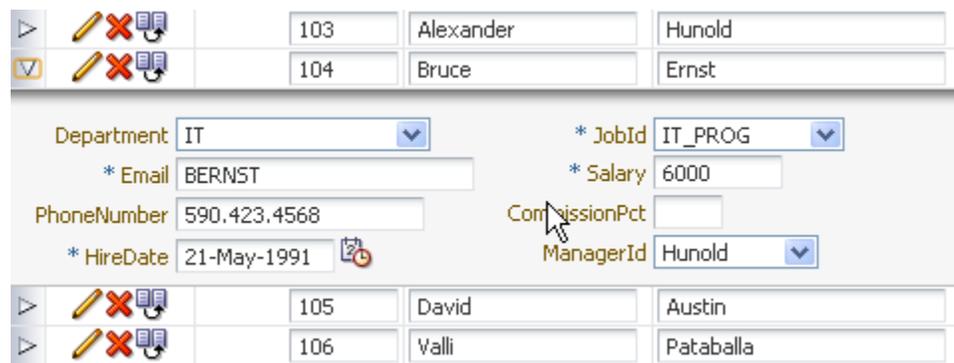
5.4.2. Using a Table Overflow Area

It is also possible to include detail information for the current row of a table within the table page. This is called table overflow. 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. You can also use table overflow to prevent horizontal scrollbars to appear when the table width exceeds the available page width.

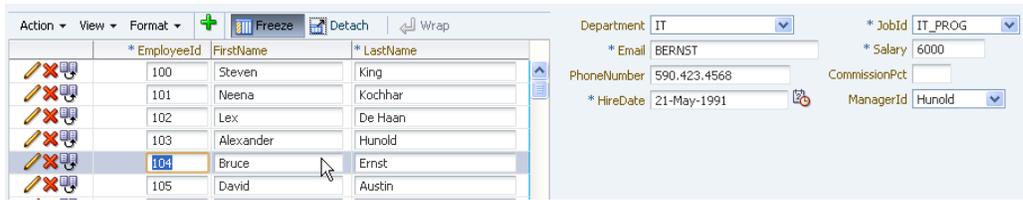
You configure a table overflow area using the **Table Overflow Style** property for the group.



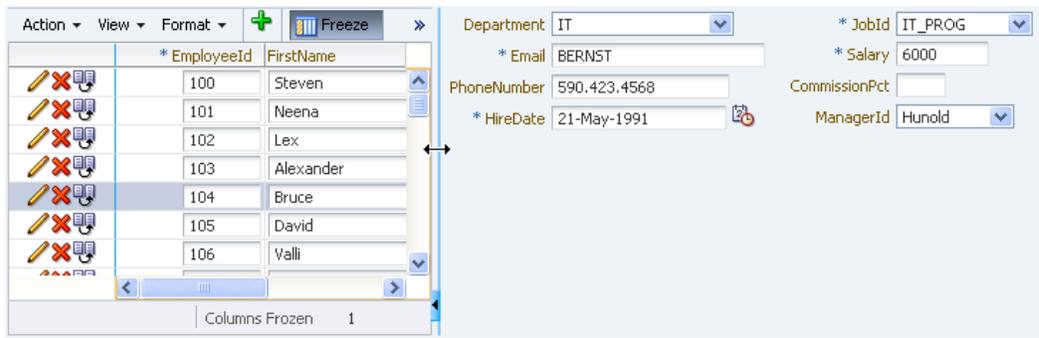
When set to *Inline*, each row in the table has an expand/collapse icon. When the user clicks expand, the overflow area is shown directly below the current row, and above the next row.



When set to `Right`, the overflow area is always displayed at the right of the table. When you change the row by clicking on it, or by using the arrow keys, the information in the overflow area will be updated for the newly selected row.



When set to `Right` with `Splitter`, the overflow area is also displayed to the right of the table, with a splitter in between. The splitter allows you to redivide the amount of horizontal space (the width) taken by the table versus the amount of space taken by the overflow area. This setting allows you to continue to use horizontal and vertical table stretching, which is not possible with overflow style `Right`. So, this setting only makes sense if you have enabled stretching as explained in the previous section [Using Table Layout and Stretching](#).



When set to `Below`, the overflow area is always shown below the table.

When set to `Below` with `Splitter`, the overflow area is displayed below the table, with a splitter in between. The splitter allows you to redivide the amount of vertical space taken by the table (the height) versus the amount of space taken by the overflow area. This setting allows you to continue to use horizontal and vertical table stretching, which is not possible with overflow style `Right`. So, this setting only makes sense if you have enabled stretching as explained in the previous section [Using Table Layout and Stretching](#).

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 [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 [Controlling Page Layout Using Region Containers, Item and Group Regions](#). Below is an example of an inline table overflow style with two item regions inside a horizontal region container.



 **Attention:** While the **Table Overflow Style** property does not have a popup style, you can generate popup windows that effectively behave as an overflow area in a popup window. See section [Generating Content in a Popup Window](#) for more information

5.4.3. Hiding Items in a Table

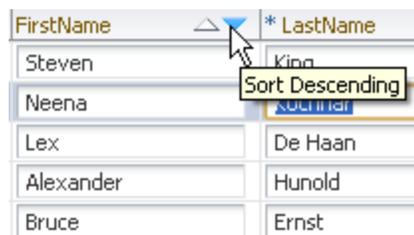
With the **Display in Table Layout?** property, you can determine which items will be generated in the table.

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

- True. The item is generated in the table page
- False. The item is not present in the generated table page
- A custom EL expression that is evaluated at runtime (must return true or false). For example, by using an EL expression, you can conditionally show/hide the item based on the value of another item, or based on task flow input parameters.

5.4.4. 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 one of the order icons that appear automatically when the user places his pointer over the column header:



To generate the sortable icons, set the **Column Sortable** property on the item level to true, for those items you wish to be sortable.

5.4.5. 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.

The screenshot shows a table with the following data:

EmployeeId	FirstName	LastName	* Salary
145	John	Russell	14000
146	Karen	Partners	13500
147	Alberto	Errazuriz	12000
148	Gerald	Cambrault	11000
149	Eleni	Zlotkey	10500
			12200

The value 12200 in the summary row is highlighted with a red box.

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.

The screenshot shows the 'Display Settings' configuration panel with the following properties:

Property	Value
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	sum
Prompt in Table Layout	average
Short prompt	count

The 'Display Summary Type in Table' dropdown menu is open, showing the options: sum, average, and count. The 'average' option is selected.

5.4.6. 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.

General

- Entity Objects
- Attributes
- Query
- Java
- View Accessors
- List UI Hints

Tuning

Enter tuning parameters for this View, to control SQL execution and t

Retrieve from the Database

All Rows
 Only up to row number

in Batches of:

As Needed
 All at Once

At Most One Row

No Rows (i.e. used only for inserting new rows)

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:

When **Use Table Range?** is true, set All Rows in Batches to the value of **Table Range Size + 1**. When **Use Table Range?** Is false or **Enable Stretching?** is checked , set it to the number of rows a user can normally see on the screen.

General

- Entity Objects
- Attributes
- Query
- Java
- View Accessors
- List UI Hints

Tuning

Enter tuning parameters for this View, to control SQL execution

Retrieve from the Database

All Rows
 Only up to row number

in Batches of:

As Needed
 All at Once

At Most One Row

No Rows (i.e. used only for inserting new rows)



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 Fusion Developers Guide, section 42.2 Tuning Your View Objects for Best Performance:

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/bcadvvo.htm#sm0342

5.5. 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 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 the selected row, or create a new row.



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 [Creating Form Pages](#) and [Creating Table Pages](#) to see which properties are available.

Steps to create a table-form page:

1. Set the **Layout Style** property to 'table-form' to generate a Table Page and a Form Page for the group.
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 a link on the item specified as **Descriptor Item**.

EmployeeId	FirstName	LastName
100	Steven	King
101	Neena	Kochhar
102	Lex	De Haan
103	Alexander	Hunold
104	Bruce	Ernst
105	David	Austin

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

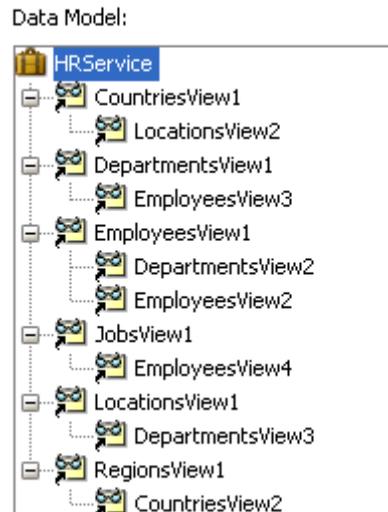
	EmployeeId	FirstName	LastName
	100	Steven	King
	101	Neena	Kochhar
 	102	Lex	De Haan
	103	Alexander	Hunold
	104	Bruce	Ernst
	105	David	Austin

Note that while not required, it is common to make the table read-only with this layout style, since editing typically occurs in the form page.

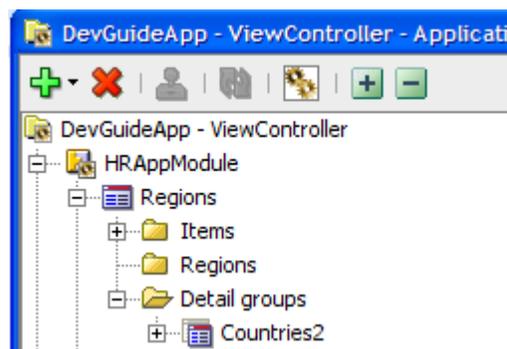
5.6. Creating Master-Detail Pages

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.



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, by default 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. If you deliberately want to choose a view object usage that is NOT nested for the detail group, you can uncheck the property **Dependent Data Collection**. When this property is unchecked, you can choose any view object usage from your data model.



5.6.1. Master-Detail on Separate Page

Uncheck the **Same Page?** property for the detail group. This indicates that the detail should be generated on a separate page.

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

On the Region (master) page, we see the following button to the detail page:



On the detail page, we see the following:



As you can see the Countries are shown on the second page.

The links on the top, so-called breadcrumbs, can be used to navigate back to the Regions table and form pages.

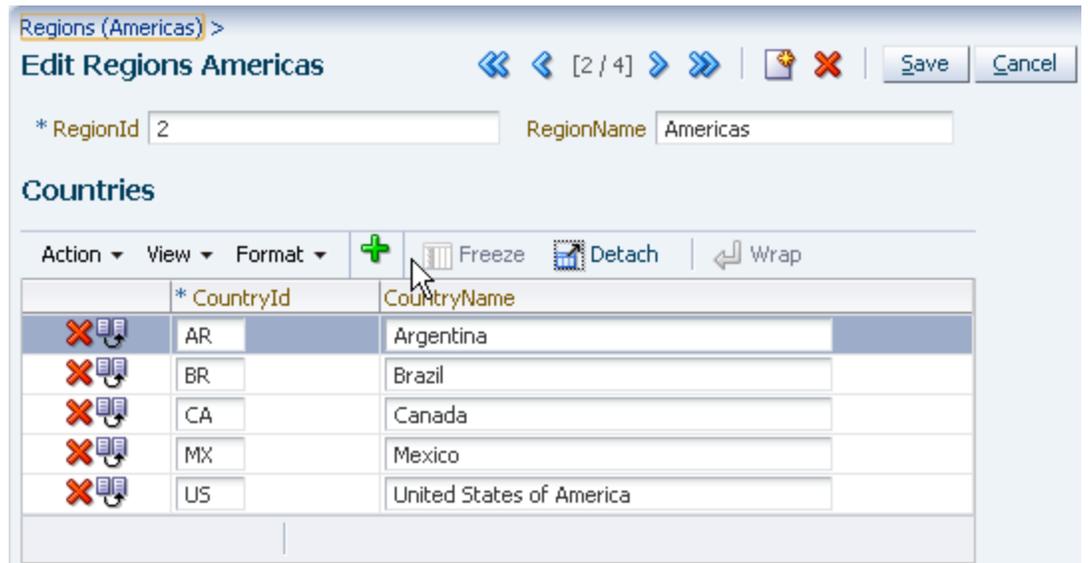
5.6.2. Master-Details on Same Page

To be able to generate master-detail groups on a single page you must check the **Same Page?** property for the detail groups you want to show on the same page as the master group. For each detail group, you can then set the **Same Page Position** property to the desired value.

Group Layout	
Layout Style *	table
Table Overflow Style	
Wizard Style Layout? *	<input type="checkbox"/>
Stack Groups on Same Page? *	None
Same Page? *	<input type="checkbox"/>
Same Page Display Position? *	Below Parent Group with Splitter
Group Width	Below Parent Group
Group Height	Below Parent Group with Splitter
Enable Stretching? *	At the Right of the Parent Group
Query Settings	At the Right of the Parent Group with Splitter
	In Table Overflow Area of Parent 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.



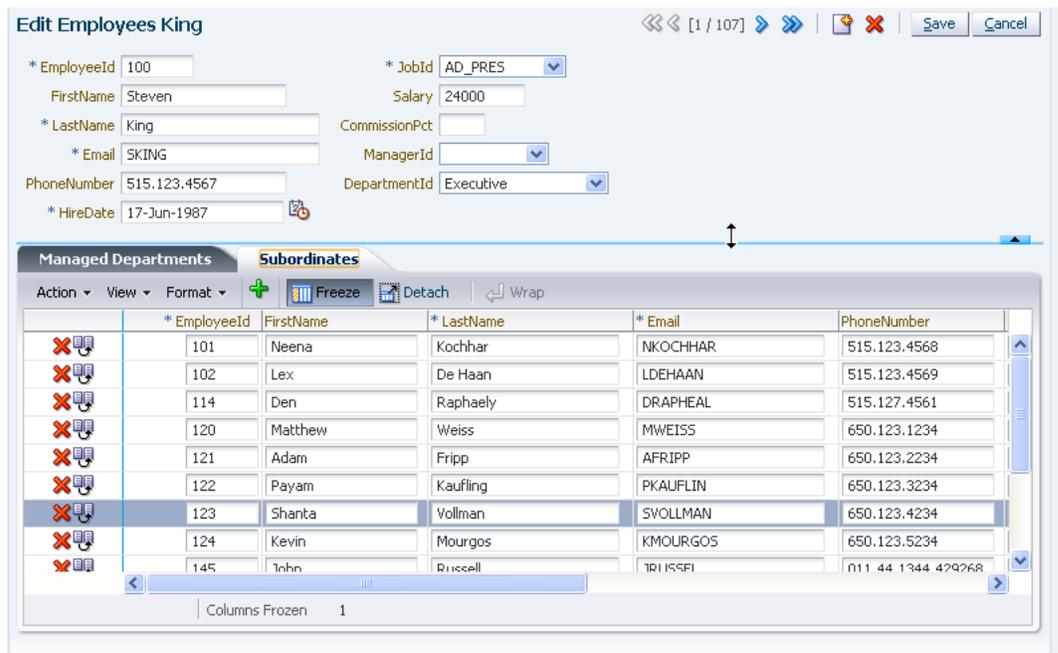
5.6.2.1. Stretching the Detail Table

If the detail group has a table layout, and you want to vertically stretch the table to show as many rows as possible, you must check the property **Enable Stretching?** on both the detail group AND the master group.

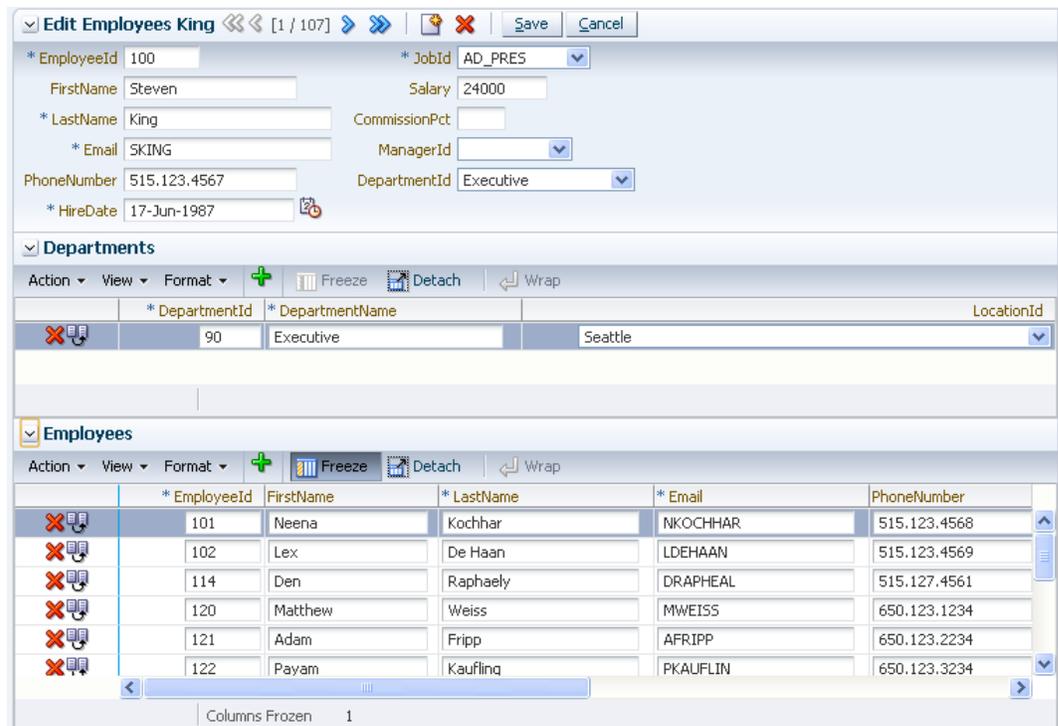
5.6.2.2. Stacking Groups on Same Page

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 using tabs or an accordion. You can achieve this by setting the **Stack Groups on Same Page** to the desired value.

In the generated screen shot below the `ManagedDepartments` and `Subordinates` groups have **Same Page** checked, and the **Same Page Display Position** set to "Below Parent Group with Splitter". The parent `Employees` group has property **Stack Groups on Same Page** set to "Detail Groups Only (Tabbed)". The initial splitter position is set using the **Group Height** property on the `Employees` group



You also stack all groups on the same page, using tabs or an accordion. The screen shot below shows the layout you get when setting the **Stack Groups on Same Page** property to All groups (Accordion).



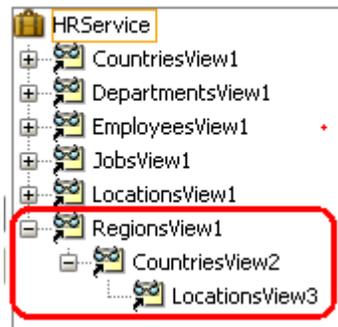
The advantage of using an accordion over tabs is that the end user can decide how many groups he wants to see simultaneously.

5.6.2.3. Grand Master-Master-Detail pages

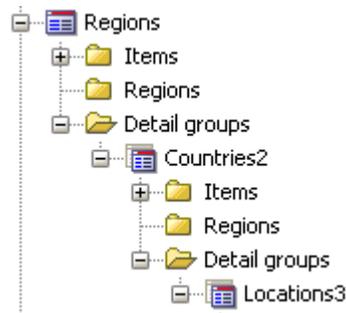
In addition to having multiple sibling detail groups, you can also have detail groups that in turn have detail groups. Again, you should first make sure the data model of the

application module supports more levels of master-detail nesting. You can nest as deeply as you want.

Data Model:



In the above screen shot we have set up three levels of nesting. In the JHeadstart Application Definition Editor, we can create the same group structure.



Again, we have multiple options to position the three groups. In the screen shot below, the Countries group has **Same Page Display Position** set to Below Parent Group while the Locations group has the same property set to At the Right of Parent group With Splitter.

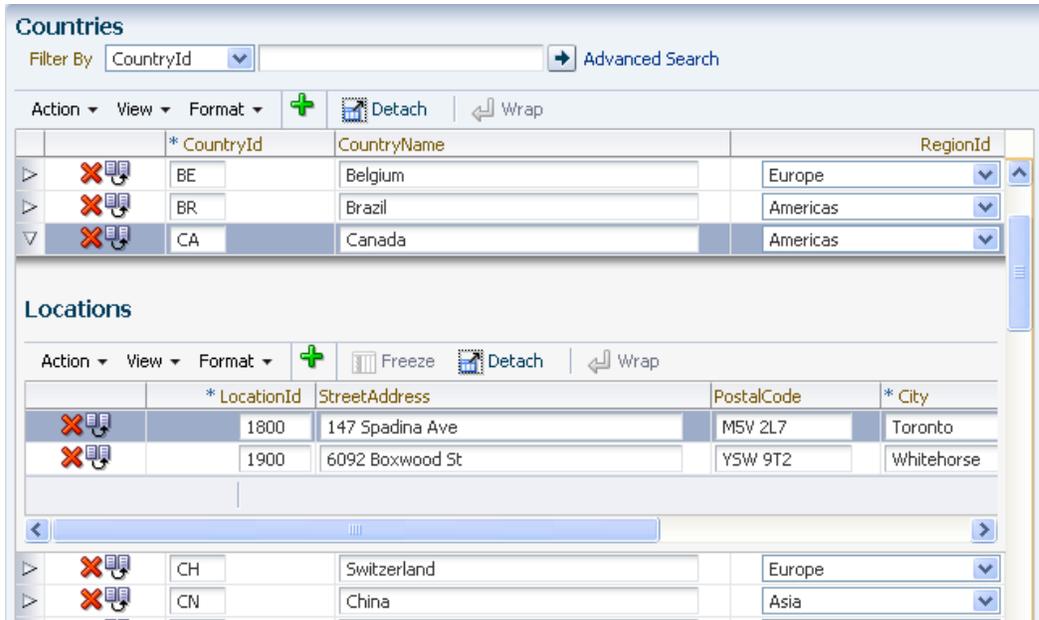
* CountryId	CountryName
AR	Argentina
BR	Brazil
CA	Canada
MX	Mexico
US	United States of America

* LocationId	StreetAddress
1800	147 Spadina Ave
1900	6092 Boxwood St

5.6.2.4. Nested Tables

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. 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.

5.6.2.5. More Advanced Layouts for Master-Detail Pages

Using Region Containers, Detail Group Regions and Item Regions, you can create even more advanced designs for your master-detail pages. For more information, see section [Controlling Page Layout Using Region Containers, Item and Group Regions](#).

5.7. 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.

5.7.1. Choosing a Tree Layout Style

There are three group layout styles that you can use to generate tree structures:

- **reusableTree**: this option only generates a reusable tree component. The tree component can be used as an alternative menu: you can link the nodes of the tree component to other groups in your application definition editor, and then the form page of this linked group will be displayed at the right of the tree, showing the row that matches the tree node that was clicked.
- **tree-form**: with this option the group generates both a tree component and associated form page that shows the row that matches the tree node that was clicked. The tree component is generated in a group-specific shell page, which means the tree-form group cannot be reused as a region within other pages. Also, the tree-form group must be a top-level group, you cannot have a detail group with tree-form layout while the parent group does not have a tree or tree-form layout.
- **tree**: this option generates a non-clickable tree node. It only makes sense to use this in combination with tree-form groups, if some of the nodes on the tree should be non-clickable. If none of the nodes in the tree should be clickable, it is better to use the reusableTree layout because you can then embed the tree in other pages.

Using the tree layout is slightly more work to configure in the application definition editor, but has the following advantages:

- The tree component is generated as a region containing a taskflow with page fragments that can be reused in other pages. The tree region contains nested regions that show the form page for each node type.
- You can reuse groups that maintain a data collection and are directly accessible through the menu. This means that you can provide various ways of accessing the same data to the end user. An end user can use the menu to navigate to a group and query a specific row, or he can use the tree component and click on the tree node to see the desired rowdata.

- When building a recursive tree, you can use the same group for showing the form page for the root nodes (the nodes where the recursive foreign key is null), as well as the child nodes. This can save a lot of (duplicate) work if other detail groups without a tree layout are shown on the same page. In the case of a tree-form layout, the two groups with tree-form layout that together make up the recursive tree, both need to have the same detail groups defined.

5.7.2. Generating a Basic Tree

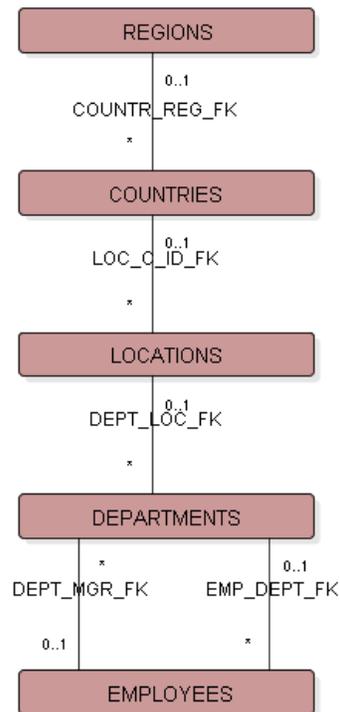
Most of the tree controls you will generate will be of the basic category. There are a few variations that will be explained in later sections:

- [Variation: Basic Tree with non-clickable 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:

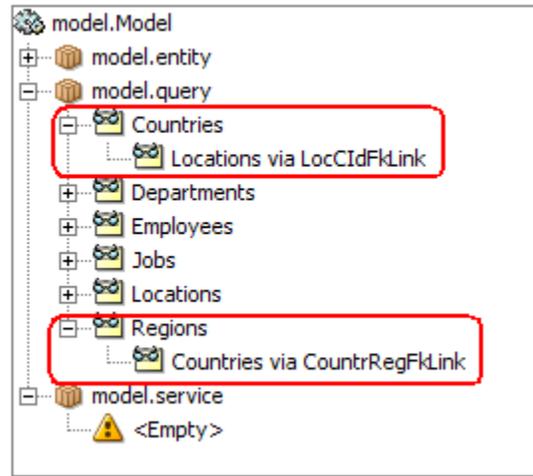


We will generate a tree with REGIONS, COUNTRIES and LOCATIONS.

Steps:

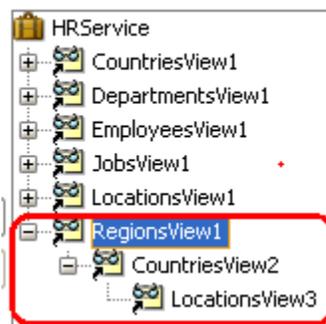
1. Check your model. Make sure you have a View Object for each level you want to show in the tree control. Check the presence of View Links between the View Objects. In this example, you will need a View Object for REGIONS, for COUNTRIES and for LOCATIONS and two View Links for the foreign keys between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

Available View Objects:



2. Add View Object Usages for these View Objects to the Data Model of the Application Module. Add Countries as a child of Regions and Locations as a child of Countries. 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:

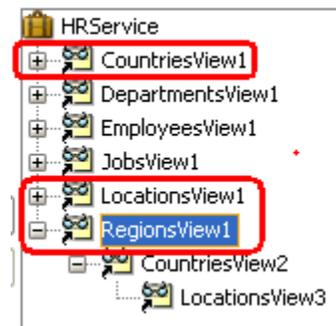
Data Model:



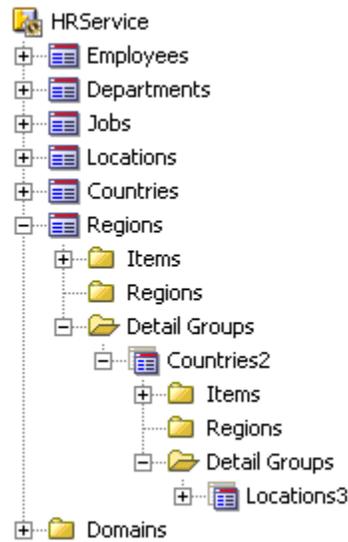
3. For each tree node that you want to be clickable, you need a top-level view object usage that can return any row of that node type. Since the RegionsView1 is already a top-level usage, you can use that usage both for the top-level tree node, as well as for

the Regions form page.

Data Model:



4. Run the 'New Service Definition' wizard based on the above application module data model. You will end up with something like this:



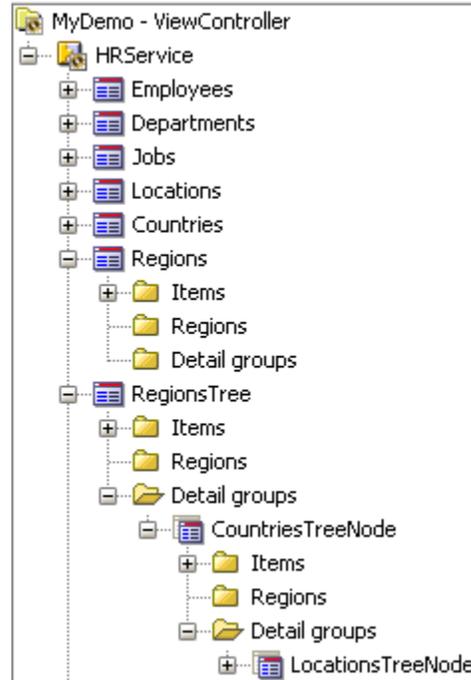
The remainder of the steps depends on whether you want to generate a reusable or non-reusable tree

5.7.2.1. Generating a Basic Reusable Tree

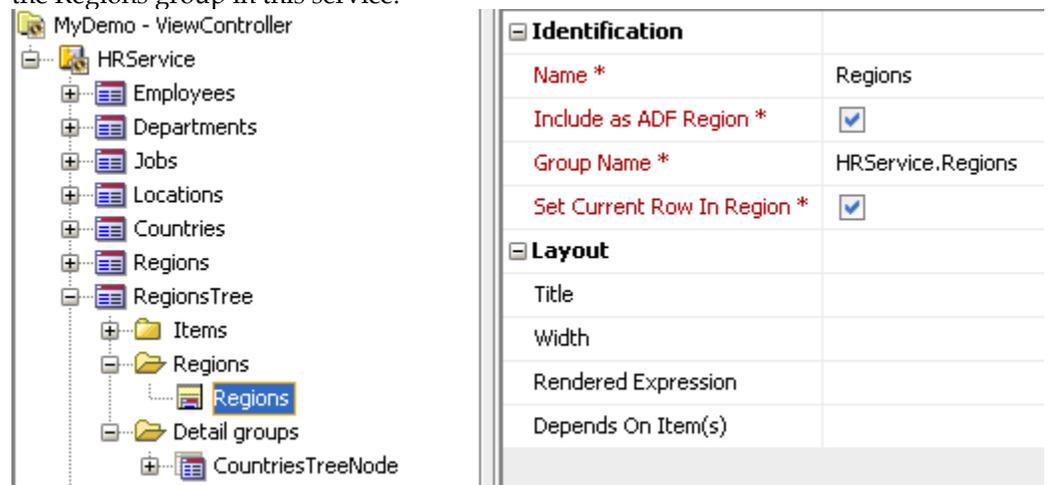
Perform the following additional steps to generate a non-reusable tree.

5. You need to make sure the JHeadstart Application Definition has groups and detail groups for generating both your tree as well as separate groups with "form" or "table-form" layout you can link to when clicking on a tree node. So, we need to make a copy of the Regions group, and rename the existing Regions group to RegionsTree. For clarity, we also rename the Countries2 group to CountriesTreeNode and the Locations3 group to LocationsTreeNode. Your group structure should look

like this:

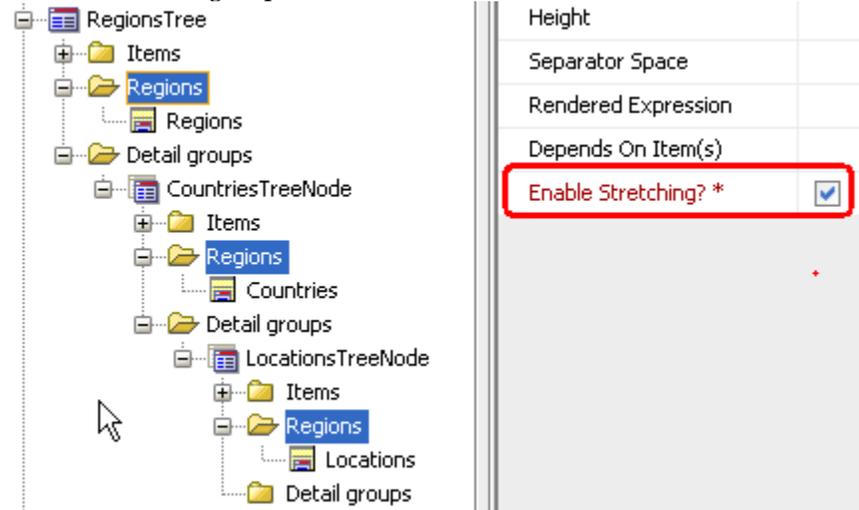


6. For the RegionsTree group and all detail groups below it, change the **Layout Style** property to 'reusableTree'.
7. For the RegionsTree group add a Group Region named "Regions", check the checkbox property **Include as ADF Region?** And set the **Group Name** property to the Regions group in this service.



8. For the CountriesTreeNode group add a Group Region named "Countries", check the checkbox property **Include as ADF Region?** And set the **Group Name** property to the Countries group in this service.
9. For the LocationsTreeNode group add a Group Region named "Locations", check the checkbox property **Include as ADF Region?** And set the **Group Name** property to the Locations group in this service.
10. For the RegionsTree group and all detail groups below it, check the checkbox property **Enable Stretching?** on each group, as well as on the "Regions" region

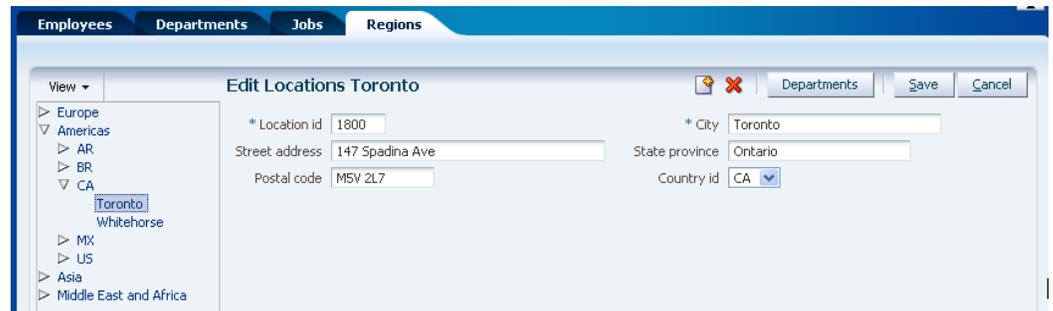
container of each group.



11. 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.
12. 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.
13. If you only want to access the "tree target" groups (Regions, Countries and Locations in the example) through the tree component, and not directly through the menu, you should switch the application definition editor to expert mode, and uncheck the checkbox **Add Menu Entry for this Group?**



That's it. Your generated tree will look similar to the picture below.



Note that the correct row is shown when clicking on a tree node because the Group Region property **Set Current Row in Region** property is checked. When checked, this property ensures that the rowKey of the selected tree node is passed as a parameter to the region (with a parameter name like rowKeyGroupName). This property is a shortcut for defining the rowkeyGroupName parameter explicitly. If you want to define the parameter explicitly, you can use `#{pageFlowScope.treeNodeRowKey}` as the value expression for the parameter.

5.7.2.2. Generating a Basic Non-Reusable Tree

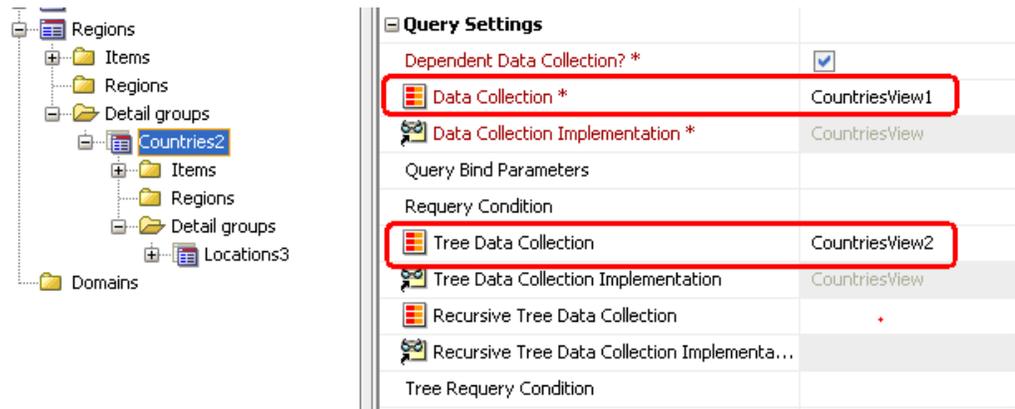
Perform the following additional steps to generate a non-reusable tree.

5. For the Regions group and all detail groups below it, change the **Layout Style** property to 'tree-form'.
6. For each of the tree-form 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 reference a separate View Object Usage at the top level in the Application Module.

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 reference a top-level View Object Usage in the Application Module.

If you originally created the groups using the New Service 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 top-level View Object Usage of that view (for example RegionsView1, CountriesView1, LocationsView1) 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 Data Collection that references a top-level View Object Usage 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. Run the JHeadstart Application Generator. You will get something like this:



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

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

5.7.3. Variation: Basic Tree with non-clickable 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 case you followed the steps for a reusable tree, you simply remove the group regions defined for the Regions and Countries group.

In case you followed the steps for a non-reusable tree, you change the **Layout Style** property of the regions and Countries2 group to 'tree'.

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

5.7.4. Variation: Recursive Tree

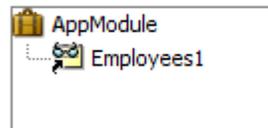
In some cases, tree structures are modeled in the database with self-referencing foreign keys (visible in Entity-Relationship diagrams by the so-called pig's ear).

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. 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.

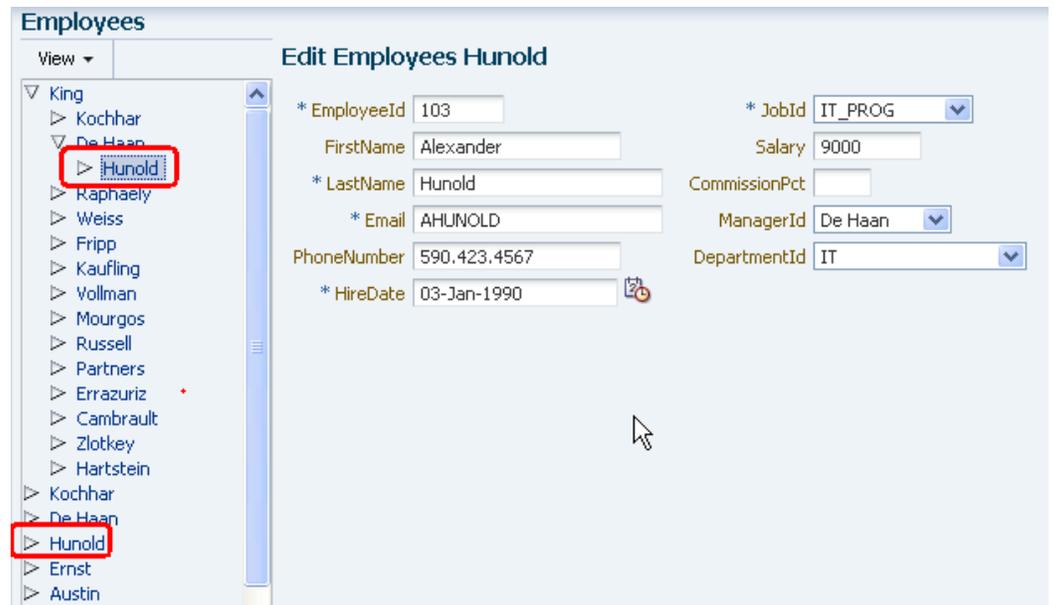
In the data model of the Application Module, it is sufficient to have only one level to generate a tree with an unlimited level of nesting. For example, to have a tree with unlimited recursion on Employees, you only need this data model:

Data Model:



Now, follow step 5 and onwards as documented in the section on the basic tree, reusable or non-reusable. The only difference is that you need to apply the steps to only one group, instead of three groups.

You will get this (nodes expanded by hand):



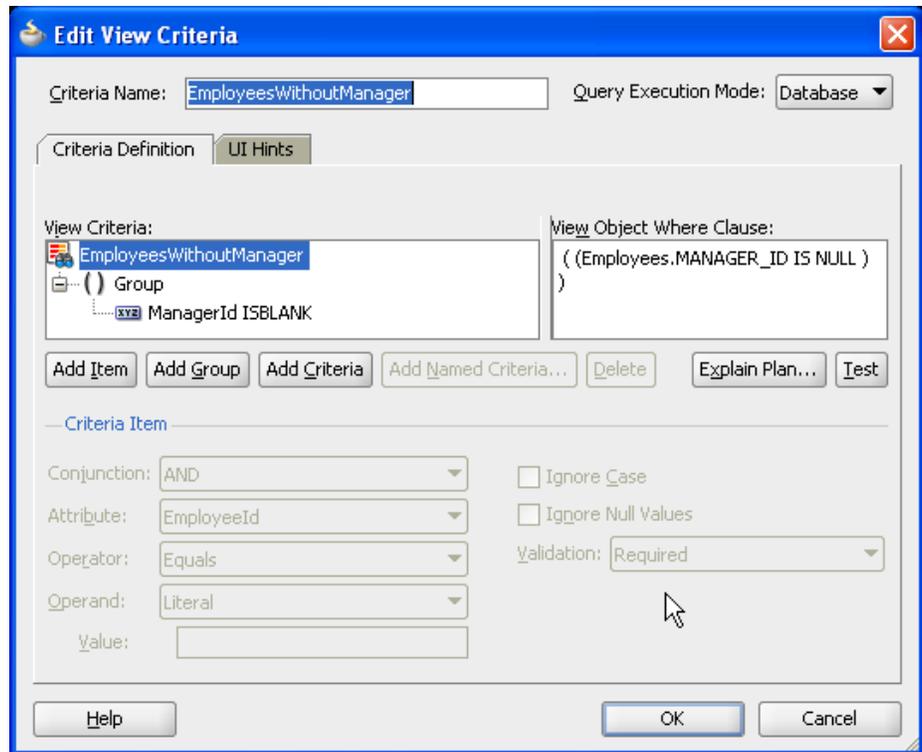
As you can see, employees that have a manager are displayed twice in the tree.

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

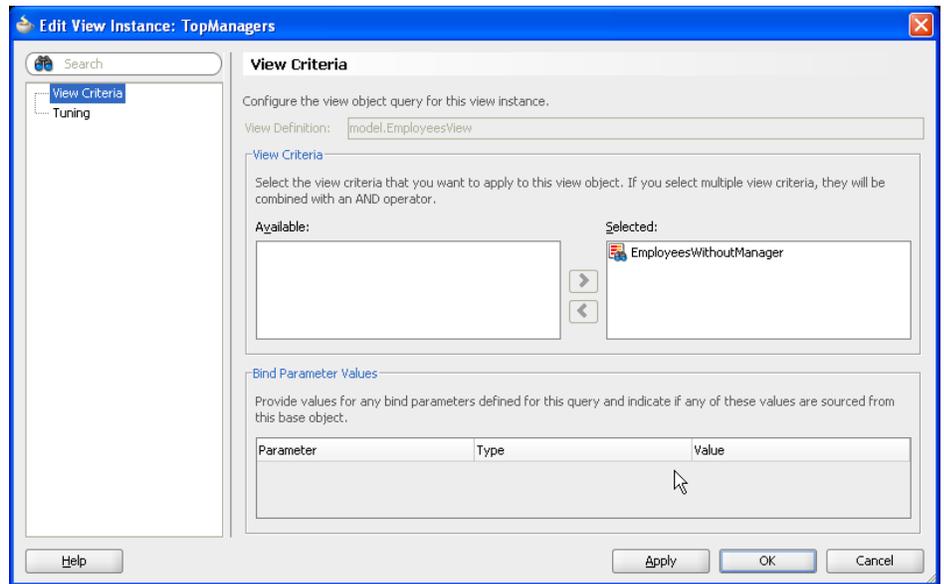
By default, every employee is displayed in the top level of the tree. In most cases, this is not what you want. In this example, you most likely want only employees without a manager to appear in the top level of the tree structure, and their subordinates below them. It is quite easy to do so by applying a named view criteria to the top-level employee view object usage, and create a nested view object usage based on the same view object.

These steps assume you have already done the steps described in section [Variation: Recursive Tree](#).

- Go to the EmployeesView view object in your Business Components project and create a new **View Criteria**.
- Name this view criteria "EmployeesWithoutManager" and add ManagerId as item with operator Is Blank.



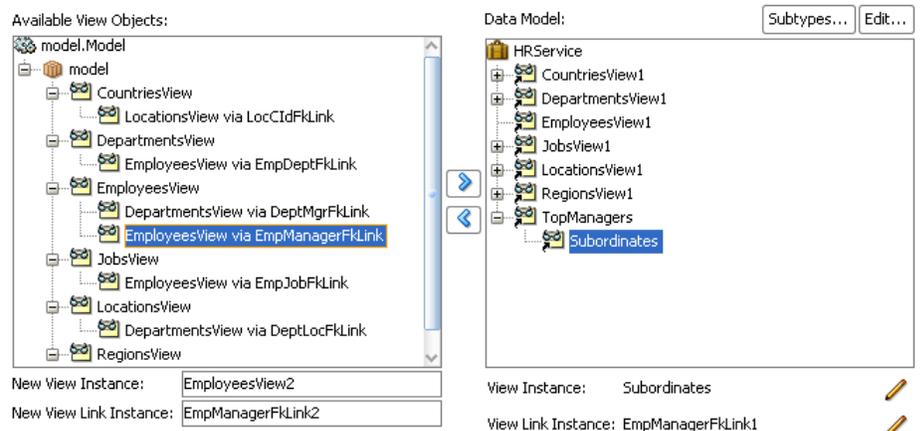
- Go to the application module data model, and create a new top-level view object usage "TopManagers" based on the EmployeesView view object. Assign the view criteria you just created to this view object usage.



- Add a nested view object usage named "Subordinates" to the 'TopManagers' view object usage, using the self referencing view link on the EmployeesView view object.

View Object Instances

The data model contains a list of view object and view link instances, displaying master-detail relationships.



5.7.5.1. Generating a reusable recursive tree

To generate a reusable recursive tree with limited set of root nodes, you need to create a master group with **Layout Style** "reusableTree" and **Data Collection** set to TopManagers1, and a detail group, also with **Layout Style** "reusableTree" and **Data Collection** set to Subordinates. Both groups have a group region that references the same Employees group. Note that region names must be unique across parent and detail groups, therefore the group region of the Subordinates group has suffix "2".

Layout Style *	reusableTree
Show Tree Expanded? *	<input type="checkbox"/>
Tree Width	
Wizard Style Layout? *	<input type="checkbox"/>
Stack Groups on Same Page? *	None
Same Page? *	<input type="checkbox"/>
Same Page Display Position? *	Below Parent Group
Group Width	
Group Height	
Enable Stretching? *	<input checked="" type="checkbox"/>
Query Settings	
Dependent Data Collection? *	<input checked="" type="checkbox"/>
Data Collection *	Subordinates
Data Collection Implementation *	EmployeesView

If you now generate again, only employees without a manager will be shown as root node. Each record is shown only once in the correct place in the tree structure.

5.7.5.2. Generating a non-reusable recursive tree

To generate a non-reusable recursive tree with limited set of root nodes, you need to create a master group EmployeesTree with **Layout Style** “tree-form” and **Data Collection** set to TopManagers1 (or EmployeesView1) and **Tree Data Collection** set to TopManagers1, and a detail group Subordinates, also with **Layout Style** “tree-form” and **Data Collection** set to EmployeesView1 and **Tree Data Collection** set to Subordinates.

Note that although the root node and child nodes both show an employee form when clicked, the generated form layout is created by two different groups. So, if you would like to display additional information for an employee, for example the departments managed by the employee, this departments detail groups must be added to both groups. This is another advantage of reusable trees: when using a reusable tree, the same group is used to generate the form layout, being the group that is referenced through the group region in both reusableTree groups.

5.7.6. Variation: Tree showing only Children of selected Parent

The steps described in this section assume that you have already built the tree as described in section [Generating a Basic Tree](#) with the reusableTree flavour.

It is **not** possible to generate a non-reusable tree as a child group.

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 region, and then for that region show a tree of the countries and locations.

With JHeadstart you can easily generate that by having a hierarchy of groups, and only setting the layout style to reusableTree for a subset of the child groups.

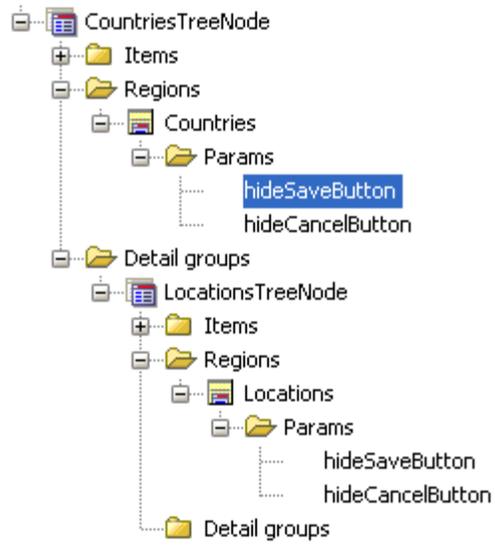
In the example used, you can simply change the layout style of the RegionsTree group to “table-form”. The detail group CountriesTreeNode is now the tree base group. In the regions table or form page, you can click the Countries button which navigates to the page fragment that shows the countries tree for the current region.



Note that you can also use **Layout Style** reusableTree on a detail group with the **Same Page** checkbox property checked. By default, you will get a layout like below.



As you can see, the Save and Cancel buttons are displayed twice. Once for the Regions master group, and once for the Locations group that is included using an ADF region. To remove the Save and Cancel buttons for the Locations (and Countries) group, you can specify additional group region parameters: hideSaveButton and hideCancelButton with value “true”.



5.8. Creating Shuttle Layouts

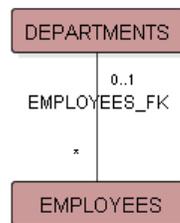
You can use JHeadstart to generate Shuttles. A shuttle is used to present a list of records to the user. The user can move records from the selected list to unselected and vice versa.

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**)

5.8.1. 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".

Group Layout	
Layout Style *	parent-shuttle
Wizard Style Layout? *	<input type="checkbox"/>
Stack Groups on Same Page *	None
Same Page? *	<input checked="" type="checkbox"/>
Same Page Display Position? *	Below Parent Group
Query Settings	
Dependent Data Collection? *	<input checked="" type="checkbox"/>
Data Collection *	Employees3
Data Collection Implementation *	Employees
Query Bind Parameters	
Requery Condition	
Domain for Unselected List in Shuttle	EmployeesShuttle
Search Settings	
Labels	
Tabname	Unassigned Employees
Display Title (Plural) *	Assign Employees to Departments
Display Title (Singular)	Employees in the Department
Show Display Title? *	<input checked="" type="checkbox"/>
Descriptor Item *	LastName

4. Generate and you will get something like this:

Edit Departments IT

* DepartmentId ManagerId

* DepartmentName LocationId

Assign Employees to Department

<p>Unassigned Employees</p> <ul style="list-style-type: none"> King Kochhar De Haan Greenberg Faviet Chen Sciarra Urman Popp Raphaely Khoo 	<p>></p> <p>>></p> <p><</p> <p><<</p>	<p>Employees in the Department</p> <ul style="list-style-type: none"> Hunold Ernst Austin Pataballa Lorentz
--	---	---

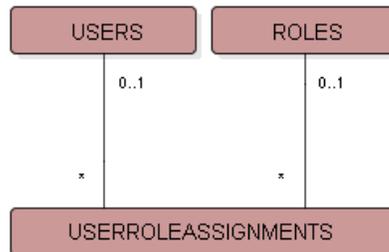


Attention: When deattaching a record (moving from right to left), the employee has no relation with any department anymore. In database terms, the department_id column is set to null. This means it is best to use a parent-shuttle with an optional foreign key.

5.8.2. Creating Intersection Shuttles

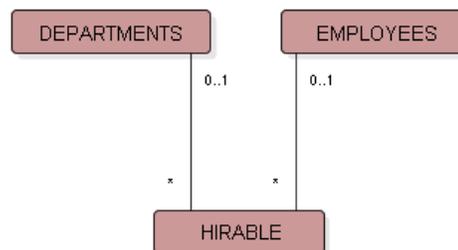
Use an intersection shuttle when you want to maintain an intersection between two ViewObjects. An intersection typically exists when there is a m:n relation between two View Objects. Examples:

- 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 hireable by multiple departments.

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



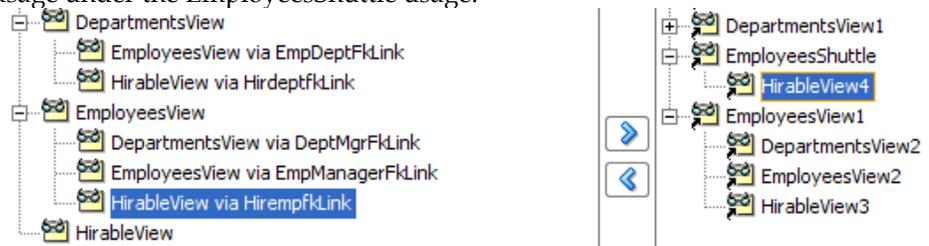
Suggestion: It is easiest to start with a new Model project and regenerate all your business components from scratch. Then you will get all necessary Associations and View Links.

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 to 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!

5.8.3. 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 queried, 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).

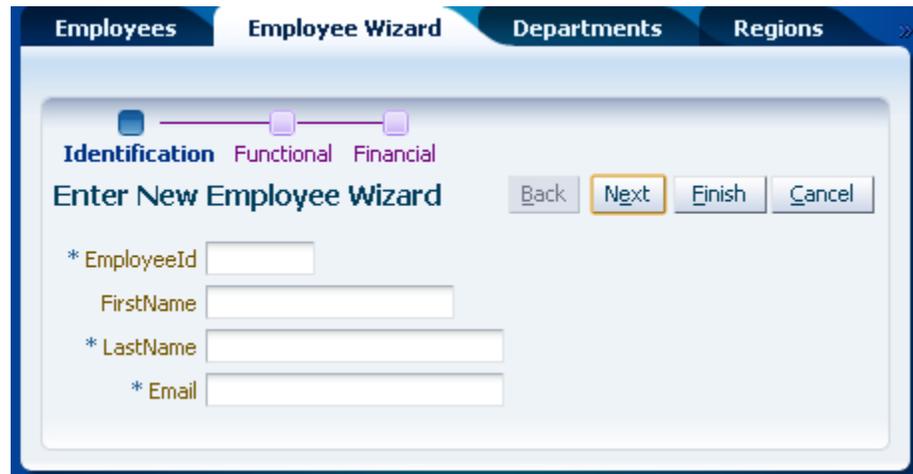
The names of relevant ViewObject usages and attributes are defined in the UI model as parameters of the Action Binding. Using EL expressions, this is also done for the leading and trailing lists submitted by the shuttle (see above).



Reference: See the Javadoc or source of `JhsApplicationModule` and `JhsApplicationModuleImpl`, in particular the methods `processParentShuttle()` and `processIntersectionShuttle()`.

5.9. Creating Wizard Layouts

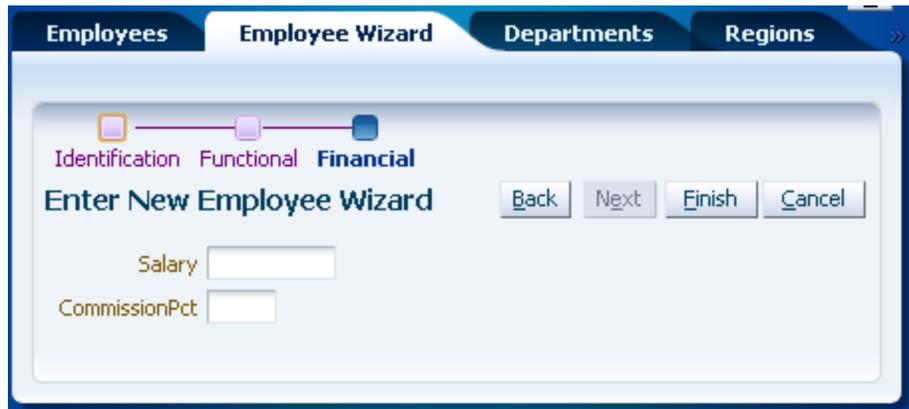
A wizard layout can be used to enter a new row in multiple steps. JHeadstart can generate item regions and detail groups to separate steps (pages) of the wizard.



To get the wizard layout, you set the **Layout Style** to `form` and check the property **Wizard Style Layout?**. To ensure that only inserts can be done in the main wizard group, we uncheck the **Single-Row Update Allowed** and **Single-Row Delete Allowed** properties. To start the wizard in create mode we set property **Display New Row on Entry** to `true`.

Group Layout	
Layout Style *	form
Wizard Style Layout? *	<input checked="" type="checkbox"/>
Stack Groups on Same Page *	None
Group Width	
Group Height	
Enable Stretching? *	<input type="checkbox"/>
Query Settings	
Search Settings	
Labels	
Operations	
Single-Row Insert allowed? *	<input checked="" type="checkbox"/>
Display New Row on Entry? *	true
Single-Row Update allowed? *	<input type="checkbox"/>
Single-Row Delete allowed? *	<input type="checkbox"/>
Model Validation Level	

To create three separate pages to insert employee data, we create three item regions as shown above, and we set the **Layout Style** of the **Regions** region container to `separatePages`.



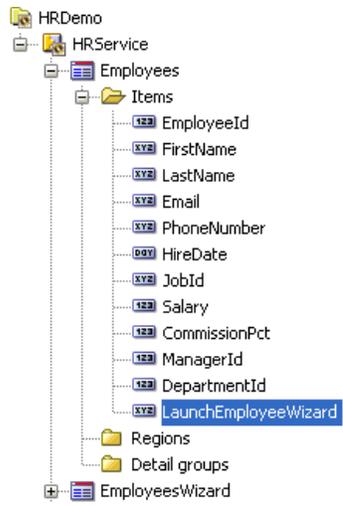
 **Attention:** Out-of-the-box the wizard style only works for creating new rows. The wizard style generation can also be used for updating existing rows. If you use table-form layout or `advancedSearch=separatePage` however, you will have to customize the Next button to perform a current row selection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

5.9.1. Launching a Wizard using New Button

When a wizard is only used to enter new rows, a common scenario is to launch the wizard group from another group based on the same data collection that has pages to maintain existing rows. You can easily configure this in the JHeadstart Application Definition editor, and also configure whether the wizard pages should replace the current page, or should be displayed in a popup window.

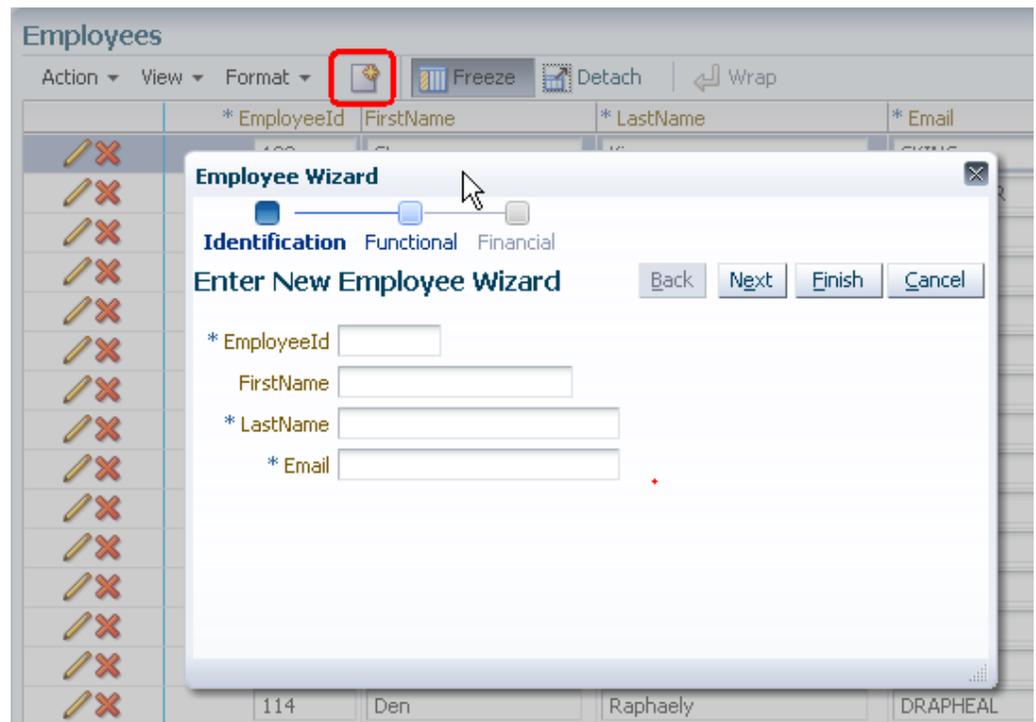
Here are the steps to do this:

- In the `Employees` maintain group, uncheck the **Multi-Row Insert Allowed** and/or **Single-Row Insert Allowed** properties to prevent generation of the standard new button to create a new row.
- Add a new unbound item called `LaunchEmployeeWizard`, and set the properties as shown below.



General	
Bound to Model Attribute? *	<input type="checkbox"/>
Name *	LaunchEmployeeWizard
Short Name	
Value	
Java Type *	String
Display Type *	groupLinkToolbarButton
Link Group Name	HRService.EmployeesWizard
Show Linked Group In	Modal Popup Window
Icon	/jheadstart/images/newRowInForm.png
Display Settings	
Display in Form Layout? *	true
Display in Table Layout? *	true
Display in Table Overflow Area? *	false
<input type="checkbox"/> Display at Right of Item	

When you now generate the application and go to the `Employees` tab, you will see an iconic new button in the table toolbar that launches the employee wizard.



Instead of setting the **Show Linked Group In** property to Modal Popup Window, you can also set it to In Page. This setting makes more sense if you click the New button in form layout.

5.10. Controlling Page Layout Using Region Containers, Item and Group Regions

This section explains how you can create arbitrary complex page layouts by using region containers, item regions and group regions.

By default each group has one empty **Region Container** named `Regions`. You can rename it as you like, but you cannot add a second top-level region container. A region container can contain three types of elements:

- Region container
- Item Region
- Group Region

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 **Group Region** can be used to create a dedicated region for a detail group (nested group) that has the **Same Page** property checked. In the section [Master-Details on Same Page](#) we have already seen that we have a number of options to layout master and detail groups on the same page by using a number of group-level properties. However, some more advanced layouts cannot be achieved using these simple group properties. By using a group region, you have full control over the relative positioning of the group.

A group region must also be used if you want to reference another top-level group that will be included as a reusable region in the page. In this case, the **Include as ADF Region** checkbox should be checked on the **Group Region**. See section [Reusing Groups](#) for an extensive example of this technique.

The **Layout Style** property of the **Region Container** determines how the child elements are laid out. It has the following allowable values:

- Horizontal
- Horizontal with Splitter
- Vertical
- Vertical with Splitter
- Tabbed
- Accordion
- Separate Pages
- Grouped Item Regions
- Modal Popup Window
- Modeless Popup Window

In the subsections below, we will provide a number of examples for the various layout styles, combined with the different element types a region container can contain.

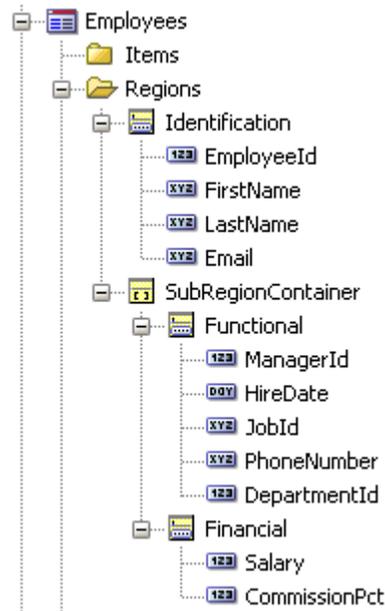
An example of using a region container with **Layout Style** `Separate Pages` is already provided in the section [Creating Wizard Layouts](#).

5.10.1. Using Item Regions

The subsections below show some examples of how you can position item groups.

5.10.1.1. Vertical and Horizontal Positioned Item Regions

Here is the structure we defined in the JHeadstart Application Definition Editor:



The **Layout Style** of the Regions region container is set to `Vertical`, the **Layout Style** of the nested `SubRegionContainer` is set to `horizontal`. The `Identification` item region has the **Columns** property set to 2, the other item regions have this property left to its default of 1. All item regions have the **Title** property set. The region containers do not have the **Title** property set. This generates a layout as shown below.

Identification	
* EmployeeId	100
* LastName	King
FirstName	Steven
* Email	SKING

Functional	
ManagerId	[Dropdown]
* HireDate	17-Jun-1987
* JobId	AD_PRES
PhoneNumber	515.123.4567
DepartmentId	Executive

Financial	
Salary	24000
CommissionPct	[Empty]

5.10.1.2. Grouped Item Regions

The structure defined in the JHeadstart Application Definition Editor is the same as in the previous section. Changes made to the properties compared to the previous section are:

- The **Layout Style** of `SubRegionContainer` is set to `Grouped Item Regions`.

- The Title of SubRegionContainer is set to Functional.
- The Functional and Financial item regions no longer have the Title property set.

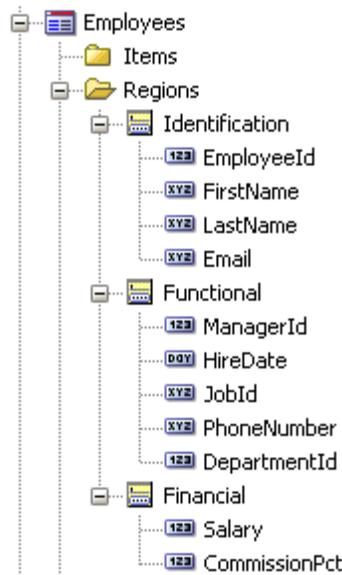
This generates a layout as shown below.

The screenshot shows a web form with two main sections. The first section, titled "Identification", contains four input fields: EmployeeId (value: 100), LastName (value: King), FirstName (value: Steven), and Email (value: SKING). The second section, titled "Functional", contains seven input fields: ManagerId (dropdown menu), HireDate (value: 17-Jun-1987), JobId (value: AD_PRES), PhoneNumber (value: 515.123.4567), DepartmentId (value: Executive), Salary (value: 24000), and CommissionPct (empty).

Note that by default an item region that does not have other elements displayed at the right, will stretch horizontally. If you do not want that to happen you can specify a number of pixels in the **Width** property.

5.10.13. Stacked Item Regions

Here is the structure we defined in the JHeadstart Application Definition Editor:



The **Layout Style** of the Regions region container is set to Tabbed. The Identification item region has the **Columns** property set to 2, the other item regions have this property left to its default of 1. All item regions have the **Title** property set. The

region containers do not have the **Title** property set. This generates a layout as shown below.

The image displays two screenshots of a web form with three tabs: Identification, Functional, and Financial. In the first screenshot, the 'Identification' tab is selected, showing fields for EmployeeId (100), LastName (King), FirstName (Steven), and Email (SKING). In the second screenshot, the 'Functional' tab is selected, showing fields for ManagerId, HireDate (17-Jun-1987), JobId (AD_PRES), PhoneNumber (515.123.4567), and DepartmentId (Executive).

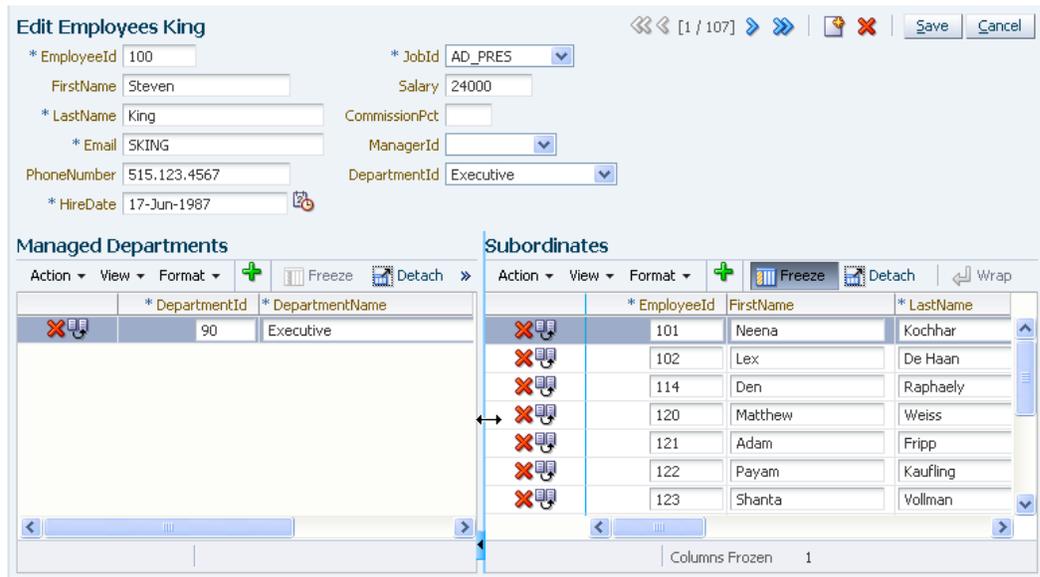
As of JDeveloper 11.1.1.4 there is a new property `dimensionsFrom="auto"` that JHeadstart will generate which ensures a clean tabbed layout without further configuration needed. It will auto-resize the tab height based on the currently selected tab when stretching is not enabled.

Note that if you are still using JDeveloper 11.1.1.3 or even 11.1.1.2, the layout of the tabbed item regions may need additional configuration by explicitly setting the **Width** and **Height** of the region container, or by checking the **Enable Stretching?** checkbox on the group and region container.

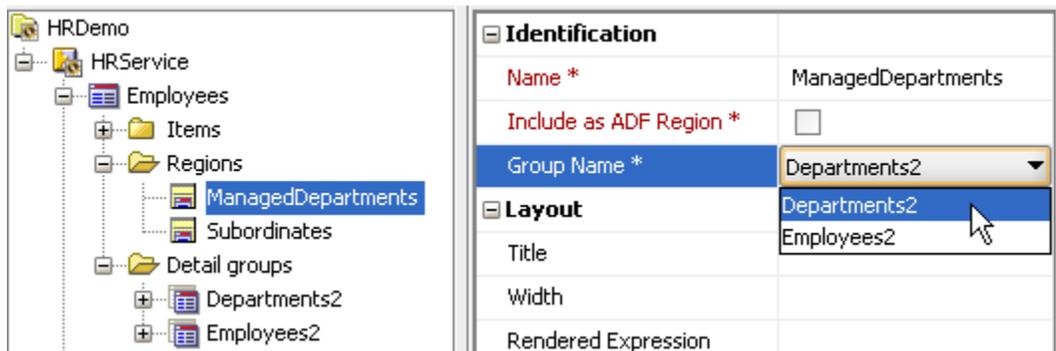
5.10.2. Using Group Regions

If the group-level properties **Same Page Display Position** and **Stack Groups on Same Page** are not sufficient to generate the required layout, you can use a combination of region containers and group regions to generate more complex layouts.

For example, if you want to generate two sibling detail groups side-by-side below the parent group, you can create a group region for each detail group and set the Regions region container **Layout Style** to `Horizontal` or `Horizontal with Splitter` as shown below.



The Employees group structure to generate the above looks as follows:

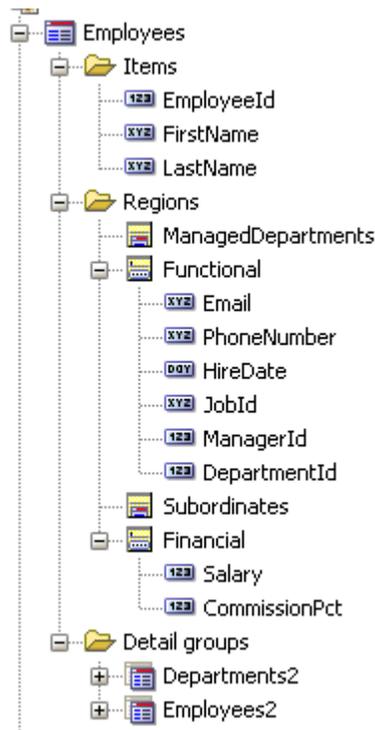


The ManagedDepartments group region has the **Group Name** set to Departments2, and the Subordinates group region has the **Group Name** property set to Employees2.

Note that you can only select detail groups in the **Group Name** property that have the **Same Page** checkbox checked, as long as the **Include as ADF Region** property is unchecked. If you check the **Include as ADF Region** property, you can only select top-level groups that are then reused and added as an ADF region to the page. You can then also define group region parameters that are passed to the group taskflow that is embedded as ADF region. See section [Reusing Groups](#) for more information.

5.10.3. Mixing Item Regions and Group Regions

Another example where you need to use group regions is when you want to mix items of the parent group with detail groups. This is often done using tabs or an accordion. In the picture below, we have set up such a structure with a Regions region container with **Layout Style** set to Tabbed, containing two group regions and two item regions.



This results in a page layout as shown below.

Edit Employees King [1 / 107] Save Cancel

* EmployeeId * LastName
 FirstName

Managed Departments Functional **Subordinates** Financial

Subordinates

Action View Format + Freeze Detach Wrap

	* EmployeeId	FirstName	* LastName	* Email	PhoneNu
	101	Neena	Kochhar	NKOCHHAR	515.123
	102	Lex	De Haan	LDEHAAN	515.123
	114	Den	Raphaely	DRAPHEAL	515.127
	120	Matthew	Weiss	MWEISS	650.123
	121	Adam	Fripp	AFRIPP	650.123
	122	Payam	Kaufling	PKAUFLIN	650.123
	123	Shanta	Vollman	SVOLLMAN	650.123
	124	Kevin	Mourgos	KMOURGOS	650.123
	145	John	Russell	JRUSSEL	011.44.
	146	Karen	Partners	KPARTNER	011.44.

Columns Frozen 1

Edit Employees King [1 / 107]

* EmployeeId * LastName
 FirstName

Managed Departments Functional **Subordinates** **Financial**

Salary
 CommissionPct

5.10.4. Generating Content in a Popup Window

When you set the **Layout Style** of a region container to `Modal Popup Window` or `Modeless Popup Window`, the child elements of this region container will be displayed in a popup window. The difference between the two layout styles is that when you choose `Modal Popup Window`, the end user cannot access the underlying page until the popup window is closed again. With a modeless popup window, the user can still use the base page while the popup window is shown.

Obviously, you need to provide a UI control to launch the popup window. JHeadstart supports the following methods to do this.

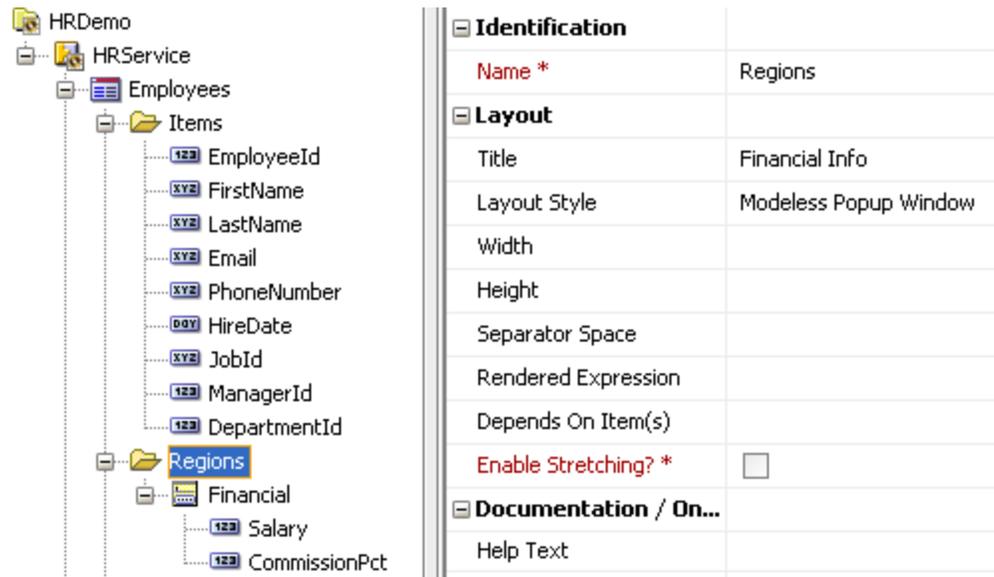
- If the **Depends on Item** property of the region container is not set, JHeadstart implicitly generates a button that launches the popup window. The label of the button is set to the **Title** property of the region container. The button will be generated at the position where normally the content of the region container would have been generated when any of the other layout styles was used.
- If the **Depends on Item** property of the region container refers to a button item, then pressing this button will open the popup window. When you use this technique, you have full control over the position of the button that launches the popup window. When you set the **Display Style** to `commandButton` you can place the button item in between any other item, or in the group toolbar by setting the **Display Type** to `toolbarButton`.
- If the **Depends on Item** property of the region container refers to a non-button item, then JHeadstart will generate a context facet for this item that launches the popup window.

Also note that by specifying a **Depends on Item** property that is displayed in table layout, the popup window is also available in table layout. This effectively allows you to implement a sort of table overflow area in a popup window while the **Table Overflow Style** property itself does not have a popup style.

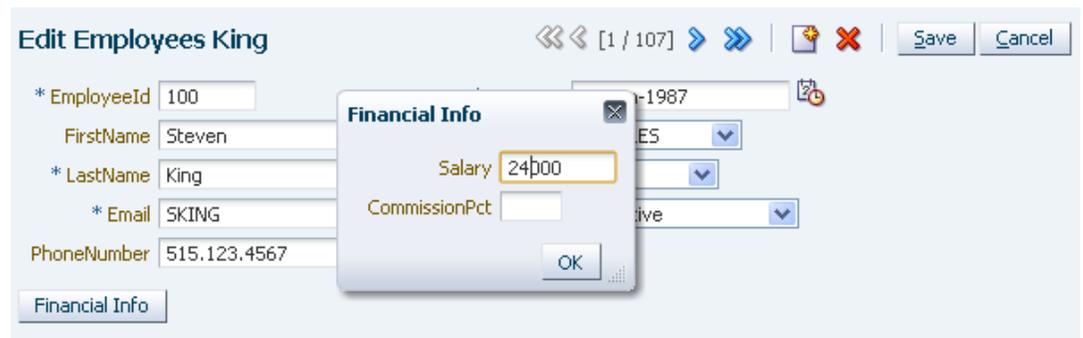
The sections below provide examples of each of the methods.

5.10.4.1. No Depends on Item property specified

In the example below, an item region with `Salary` and `CommissionPct` items is defined inside the region container. No **Depends on Item** property is specified for the region container.



When we generate the page, the `Financial Info` button is generated below the items not contained by an item region.



Note that if you specified the popup as a modeless window, you can still browse through the underlying rows, and the popup information will be updated accordingly.

5.10.4.2. Depends on Item property Refers to Button Item

In the example below, an unbound button item `ShowFinancialInfo` has been added, and the `Regions` region container has the **Depends on Item** property set to this item.

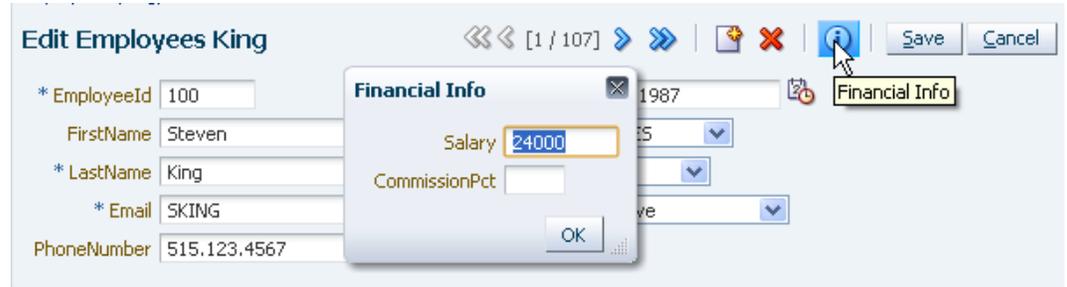
General	
Bound to Model Attribute? *	<input type="checkbox"/>
Name *	ShowFinancialInfo
Short Name	
Value	
Java Type *	String
Display Type *	commandButton
Action	
Action Listener	
Method Call	
Icon	
Display Settings	
Display in Form Layout? *	true
Display in Table Layout? *	true
Display in Table Overflow Area? *	false

When we generate the page, the Financial Info button is now generated in between the other items.

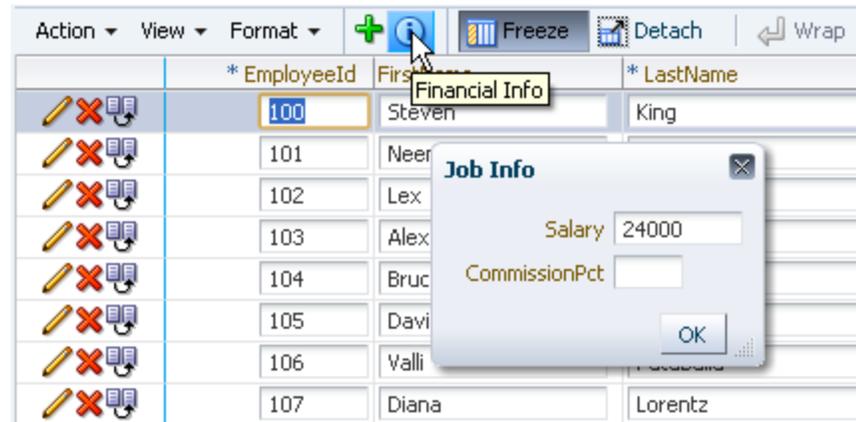
We can also change the **Display Type** to `toolbarButton` and specify the **Icon** property.

General	
Bound to Model Attribute? *	<input type="checkbox"/>
Name *	ShowFinancialInfo
Short Name	
Value	
Java Type *	String
Display Type *	toolbarButton
Action	
Action Listener	
Method Call	
Icon	/jheadstart/images/info.png
Display Settings	
Display in Form Layout? *	true
Display in Table Layout? *	true

When we now generate the page, the `Financial Info` iconic button is generated in the group toolbar.



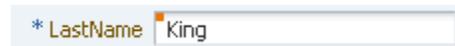
Since the `ShowFinancialInfo` item also has **Display in Table Layout?** Set to true, the item is also generated in the table toolbar.



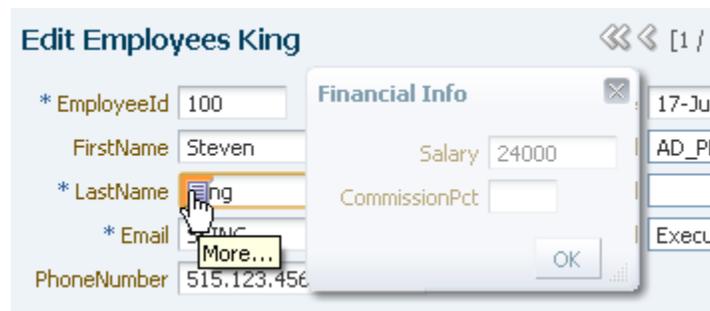
And just like in form layout, if the window is modal and you click on another row, or use the arrow keys to browse through the table rows, the information in the popup window will be refreshed accordingly.

5.10.4.3. Depends on Item property Refers to Non-Button Item

In the screen shots below, the `Regions` region container has the **Depends on Item** property set to `LastName`. `JHeadstart` then generates a `context` facet for the item with an `af:contextInfo` element inside it that launches the popup. Because a context facet is generated, ADF Faces renders a little orange square in the upper left corner of the item.



If you hover the mouse over the orange square, a tool tip "More . . ." is displayed and when you click on it, the popup is launched.



5.10.4.4. Showing Lookup Information from Another Group in Popup Window

You can also use any of the three methods discussed above to show detailed lookup information in a popup that is provided by another group in the JHeadstart Application Definition Editor. For example, on the `JobId` item in the `Employees` group, we want to provide the option to quickly look up all the details of the job, including the `JobTitle`, `MinSalary` and `MaxSalary`, as shown below.



This page has been generated with the following settings:

Identification	
Name *	Regions
Layout	
Title	Job Info
Layout Style	Modeless Popup Window
Width	400px
Height	
Separator Space	
Rendered Expression	
Depends On Item(s)	JobId
Enable Stretching? *	<input checked="" type="checkbox"/>

- The `Regions` region container has the settings as shown above.
- The `JobInfo` group region has the **Include as ADF Region** checkbox checked and has the **Group Name** property set to `Jobs`.
- Two parameters are specified to pass the current `JobId` and to show the `Jobs` page in read-only mode
- The `readOnlyMode` parameter can be set because it is also defined as parameter against the `Jobs` group.
- The **Insert Allowed**, **Update Allowed** and **Delete Allowed** expressions in the `Jobs` group all have the value `# { !pageFlowScope.readOnlyMode }`.

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.

☐ Display Settings	
Display in Form Layout? *	true
Display in Table Layout? *	true
Display in Table Overflow Area? *	false
Prompt in Form Layout	EmployeeId
Prompt in Table Layout	

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:

☐ Query Settings	
Include in Quick Search? *	<input checked="" type="checkbox"/>
Include in Advanced Search? *	<input checked="" type="checkbox"/>
Prompt in Search Region	

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.

Enter New Employees

* Employee Id	<input type="text"/>	* Job Id	<input type="text"/>
First Name	<input type="text"/>	Salary	<input type="text" value="1000"/>
* Last Name	<input type="text"/>	Commission Pct	<input type="text"/>
* Email	<input type="text"/>	* Manager Id	<input type="text"/>
Phone Number	<input type="text"/>	* Department Id	<input type="text"/>
* Hire Date	<input type="text"/>		

6.2.1. Using EL expressions

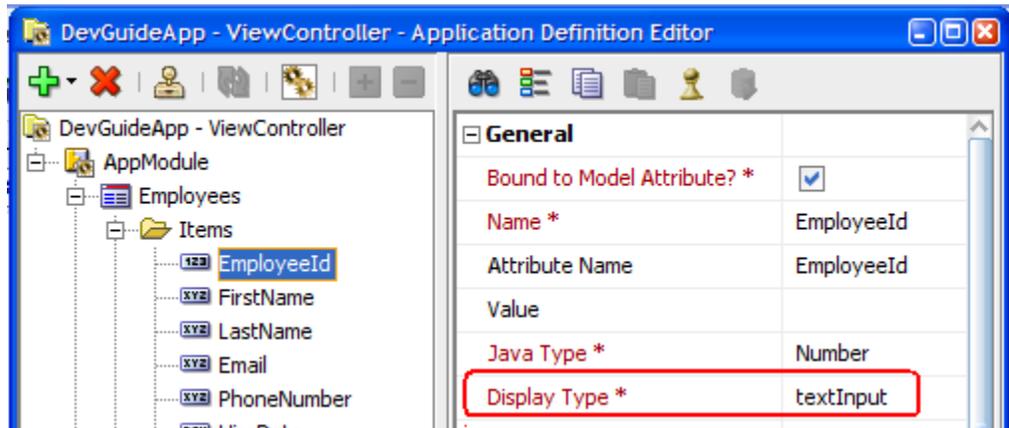
In addition to literal values, you can enter Expression Language in the **Default Display Value**.

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.

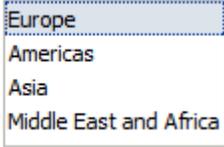
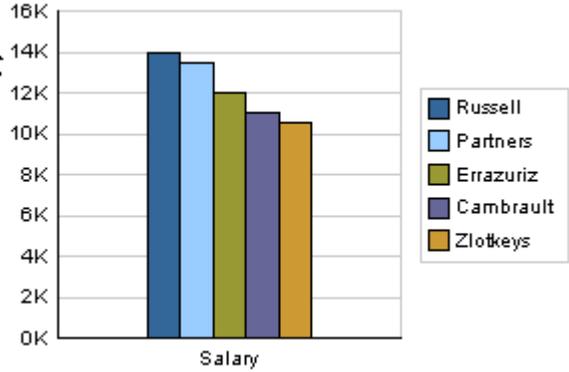
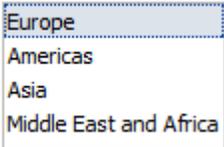
6.3. Display Type

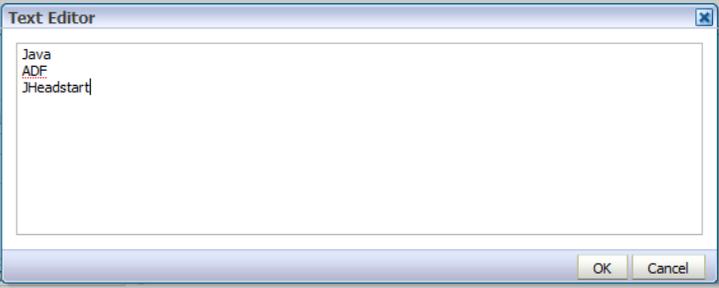
Default display types are set during the creation of the Application Definition. These can be overridden by using the **Display Type** property.



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

textInput	Phone Number <input type="text" value="650.507.9844"/>
dropDownList	Region Id <input type="text" value="Europe"/> <ul style="list-style-type: none"> Europe Americas Asia Middle East and Africa
lov	Department Name <input type="text" value="IT Helpdesk"/>
model-choiceList model-comboBox	Region Id <input type="text" value="Europe"/> <ul style="list-style-type: none"> Europe Americas Asia Middle East and Africa
model-comboBoxLov	* Region <input type="text" value="2"/> <ul style="list-style-type: none"> Europe Americas Asia Middle East and Africa <input type="text" value="Search..."/>
model-inputTextLov	* Region <input type="text" value="Europe"/>

model-listBox	* Region 
model-radio-horizontal	* Region <input checked="" type="radio"/> Europe <input type="radio"/> Americas <input type="radio"/> Asia <input type="radio"/> Middle East and Africa
model-radio-vertical	* Region <input checked="" type="radio"/> Europe <input type="radio"/> Americas <input type="radio"/> Asia <input type="radio"/> Middle East and Africa
checkbox	Lease Car? <input type="checkbox"/>
graph	
list	Region 
radio-vertical	Region <input checked="" type="radio"/> Europe <input type="radio"/> Americas <input type="radio"/> Asia <input type="radio"/> Middle East and Africa
radio-horizontal	Region <input checked="" type="radio"/> Europe <input type="radio"/> Americas <input type="radio"/> Asia <input type="radio"/> Middle East and Africa
editor	Required Skills  +

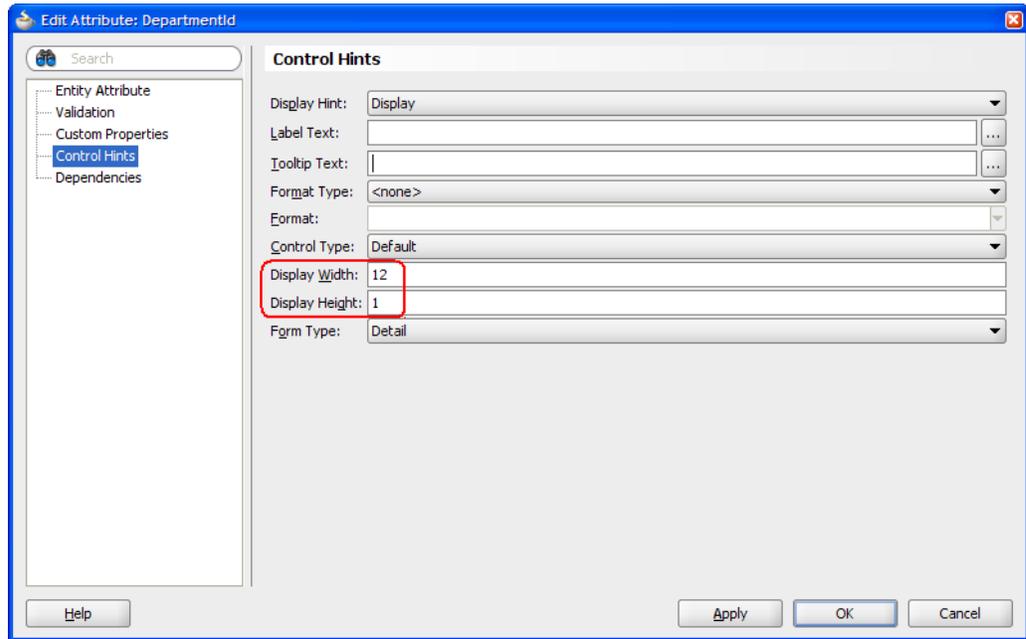
	
dateField	* Hire Date <input type="text" value="10-Jun-1999"/> 
datetimeField	* Delivery Date <input type="text" value="10-Jun-1999 00:00"/> 
secret	* Password <input type="password" value="•••••"/>
fileUpload	<p>Photo myphoto.gif <input type="button" value="Update..."/></p> <p>+</p> 
fileDownload	<p>Contract Contract </p> <p>Audio </p>
image	<p>Photo </p>
hidden	
commandButton, commandToolBar Button	<p><input type="button" value="Pay Now"/></p> 

groupLink, groupLinkButton	
FlexRegion	See Chapter 13 "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 dynamic. EL expressions are used to get the width and height from the underlying ADF Business Components. When these properties are not set on the View Object, the Entity 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? *	false
Display in Table Overflow Area? *	true
<input checked="" type="checkbox"/> Display at Right of Item	
Display Summary Type in Table	
Prompt in Form Layout	#{HINTS\$.label}
Prompt in Table Layout	
Short prompt	
Width	#{HINTS\$.displayWidth}
Height	#{HINTS\$.displayHeight}

Note that \$HINTS\$ is a special token replaced by JHeadstart upon generation. This is done because the actual EL expression to get a UI hint for items in a table layout is different from items displayed in form layout.

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.

Width	<code>#{HINTS\$.displayWidth}</code>
Height	<code>#{HINTS\$.displayHeight}</code>
Maximum Length	4

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.



Suggestion: It is also possible to use ADF Model-based lists and set the display type to 'model-choice' to get the same result for the end user. See section [ADF Model LOV vs. Custom JHeadstart LOV](#) for more information.

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.

General	
Bound to Model Attribute?	<input checked="" type="checkbox"/>
Name *	StateProvince
Attribute Name	StateProvince
Value	
Java Type *	String
Display Type *	dropDownList
Domain	StatesOfAmerica

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 the 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.⁷

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 *	RegionDomain
Domain Type *	Dynamic
[-] Query Settings	
Data Collection	Regions
Data Collection Implementation	Regions
Dynamic Data Collection Expression	
Query Bind Parameters	
Data Collection Changes By Row? *	<input type="checkbox"/>
Requery Condition	
[-] Display Settings	
Value Attribute	RegionId
Meaning Attribute	RegionName

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? *	<input checked="" type="checkbox"/>
Name *	RegionId
Attribute Name	RegionId
Value	
Java Type *	Number
Display Type *	dropDownList
Domain	RegionDomain

See chapter 7, section 7.2.2 “Using Query Bind Variables” for an example where the content of a drop down list changes dynamically based on query bind variables.

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.



Suggestion: It is also possible to use ADF Model based list and set the display type to 'model-radio' to get the same result for the end user. See section 6.7.2 for more information.

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.

☐ General	
Bound to Model Attribute?	<input checked="" type="checkbox"/>
Name *	ManagerId
Attribute Name	ManagerId
Value	
Java Type *	Number
Display Type *	radio-vertical
Domain	EmployeesViewLookup

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 the 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.

6.7.1. ADF Model LOV vs. Custom JHeadstart LOV

JHeadstart can generate two different kinds of LOV items:

6.7.1.1. ADF Model based LOV

ADF Business Components allows a user to define a List of Values on an attribute in their View Object.



Fusion Dev Guide, section 5.11 “Working with List of Values (LOV) in View Object Attributes”. Includes instructions on defining LOV in the ADF View Object itself.

http://download.oracle.com/docs/cd/E12839_01/web.1111/b31974/bcquerying.htm#CHDHBDDE

Some ADF Faces components can use this information from Business Components. There are now components like `<af:inputListOfValues>` that can be bound directly to the LOV defined in ADF Business Components and deliver full LOV functionality. Those components do not only support a ‘traditional’ List of Values (popup with search functionality), but also dropdown lists, radio groups, list boxes and even a clever combination of a dropdown with commonly used items plus an extra popup LOV for less commonly used items (called `ComboBoxLov`). See section 6.3 [Display Type](#) for some examples.

ADF Model LOV works fine for standard requirements regarding List of Values. However, the entire popup window that belongs to the LOV is generated by the component itself. This leaves little coding to the end user (smaller files, faster development), but it poses some limitations:

1. It is not possible to use multi-select functionality (where one row in the base view object will be created per selected item).
2. It is not possible to use ‘detail disclosure’ inside the LOV: displaying extra attributes for a row when the user clicks on that row.
3. It is not possible to change the layout of the List of Values (except for standard skinning possibilities), or add any extra content.

When these limitations are no problem for your LOV, the standard ADF Model LOV will work just fine. The advantage of such an LOV is also that the new model-based query component (see chapter 7) can only use these LOV's (not the custom JHeadstart LOV).

6.7.1.2. Custom JHeadstart LOV

This approach creates a List of Values by generating a custom set of components (text field, LOV icon, popup, beans, taskflow, etc.) that also implement LOV behavior. Also the page that is displayed inside the List of Values is completely generated by JHeadstart.

The end result is therefore (much) more code compared to an ADF Model based LOV, but the limitations of the ADF Model LOV do not apply. You have full control over the layout and behavior of the LOV.

6.7.1.3. Searching and LOVs

There is another dimension to this discussion: search areas. As put forward in chapter 7, there are two different kinds of search areas:

1. **ADF Model based** quick/advanced search. This approach uses information out of Business Components to support a full quick/advanced search area with a single UIComponent (<af:query> or <af:quickQuery>). Especially the advanced search area options are quite extensive.
2. **Custom JHeadstart** quick/advanced search. Like custom LOVs, this approach generates more code, but customization options are better.

The issue here is: the search and LOV approaches don't mix well. Here is a compatibility matrix:

	ADF Model search	JHeadstart search
ADF Model LOV	Supported	Not supported. You will get a text field instead of an LOV. Hide this field in the search area and use a secondary field, with a custom JHeadstart LOV, to overcome this situation.
JHeadstart LOV	Not supported. You will get a text field instead of an LOV; unless you define a Model LOV on your base view object attribute as well.	Supported

Unless there are strong reasons to 'mix' the two approaches, this is not recommended. Maintaining double LOV definitions can be tedious, besides they will be slightly different in terms of look & feel and functionality.



Reference: For more information about ADF Model search vs. custom JHeadstart search, see chapter 7: "Generating Query Behaviors".

6.7.1.4. How to choose?

As we have seen, both approaches have their pros and cons.

In general, reasons to choose the ADF Model approach include:

- There are no specific requirements that can't be implemented.
- ADF Model search must be used
- Generated code should be as compact and easy as possible

Custom JHeadstart LOV elements supply more features and customization possibilities, but may also be a bit harder to understand. Reasons to use them include:

- There are (some) customization requirements
- JHeadstart search must be used
- Special features are needed: multi-select LOV, detail disclosure or full control over the layout in the search area in the LOV.

6.7.2. Creating an ADF Model LOV

Most work in creating an ADF Model LOV is to create the definition of the List of Values in the View Object itself. See the Fusion Dev Guide how to do this. Note that List of Values is actually a bit of a confusing terminology here, because such a list could also be a dropdown list or even a radio group, not necessarily a popup window with a separate List of Values.



Reference: Fusion Developer Guide for Oracle Application Development Framework, section 5.12 “Working with List of Values (LOV) in View Object Attributes”. Includes instructions on defining LOV in the ADF View Object itself.

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/bcquerying.htm#CHDHB DDE

Once you have created your definition, you only have to set the correct display type for that attribute in JHeadstart. An overview of available display types (they all begin with “model-“) are in section 6.3.

Note that the display type set in JHeadstart and the ‘Default List Type’ in the View Object LOV editor have to match; otherwise you will get a warning. Fortunately, if you use the New Service Definition Wizard or the item synchronizer, you will get the correct display type (as defined in the Model LOV) automatically.

6.7.3. Creating a custom JHeadstart LOV

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

6.7.4. 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. Set the **Group Usage** to 'List of Values Window'.

[-] Identification	
Name *	RegionLookup
Short Name	
Description	
Bound to Model Data Collection? *	<input checked="" type="checkbox"/>
Group Usage	List of Values Window
Group Region Access	
Allow Multiple Selection in LOV? *	<input type="checkbox"/>
Use in Dialog Window? *	<input type="checkbox"/>
Group Image / Icon	
[-] Group Layout	
Layout Style *	table
Table Overflow Style	

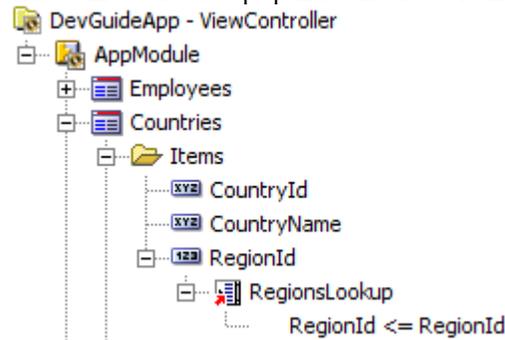
4. Specify at least one type of search for the LOV group: quick search or advanced search.



Improvement: In JDeveloper 10.1.3 there were some extra steps needed when you based your LOV on a read-only View Object (checking primary key attributes, adding setManagerKeyByRows). This is no longer necessary in JDeveloper 11.

6.7.5. Linking a Reusable LOV group to an item

1. Set the **Display Type** of the LOV item (which will have a LOV attached) to 'lov'.
2. Select the LOV item and press the green plus (+) symbol.
3. 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).

* Region 

6.7.6. 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.

Edit Locations Venice

* Location Id

Street Address

Postal Code

* City

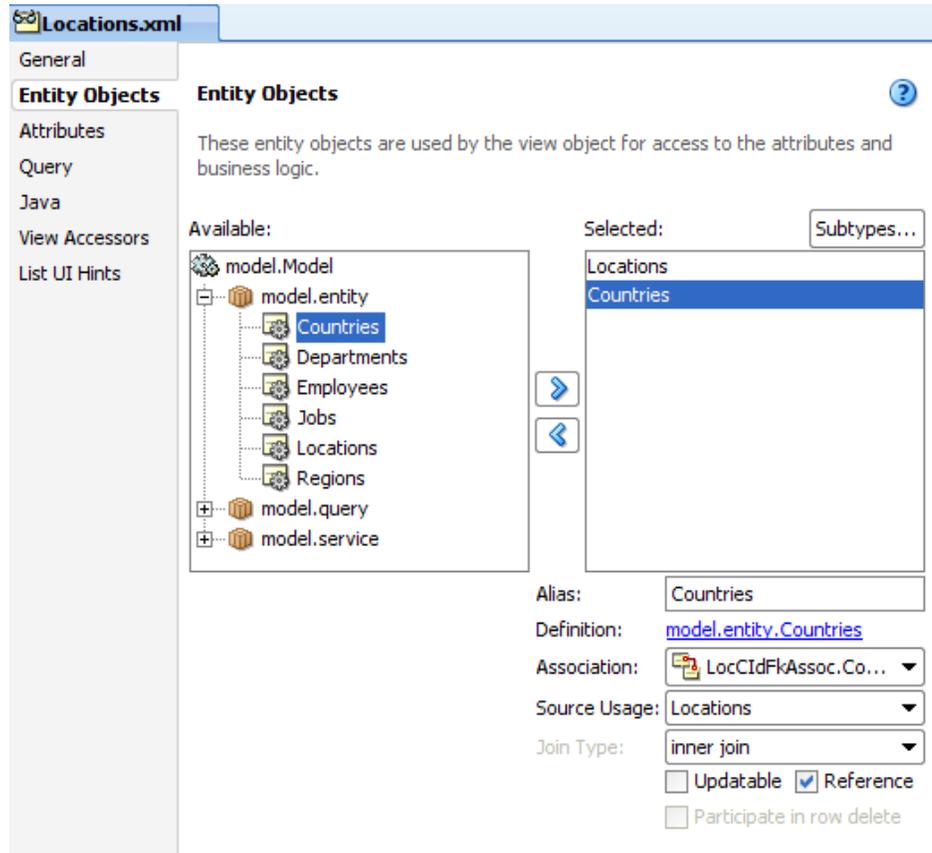
State Province

* Country Id 

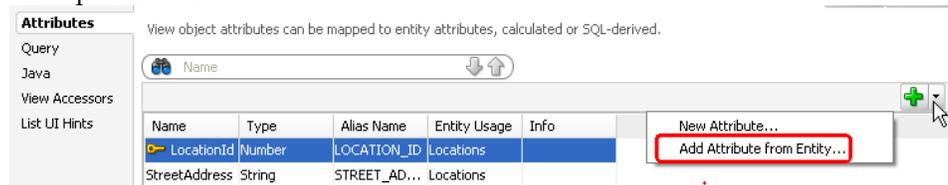
In that case you must include all the attributes you want to display on the page to become a part of the view object on which you base your group in the Application Definition. So the CountryName attribute of the Countries View must be added to the LocationsView.

Perform the following steps to accomplish this:

1. Double click the view object you want to modify
2. Navigate to the Entity Objects tab. 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, click the button 'Add from Entity' on the top of the screen.



5. 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. Click OK.
6. 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. Right-click on the attribute and select 'Rename'. Rename the Id1 attribute using the naming convention LkpXxxId to avoid confusion.
7. 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.**

- 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.

Name	Type	Column	Info
 LocationId	Number	LOCATION_ID (L...	
StreetAddress	String	STREET_ADDRES...	
PostalCode	String	POSTAL_CODE (L...	
City	String	CITY (Locations)	
StateProvince	String	STATE_PROVINC...	
CountryId	String	COUNTRY_ID (Lo...	
LkpCtrCountryN...	String	COUNTRY_NAME ...	
LkpCtrCountryId	String	COUNTRY_ID1 (C...	

- 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.

6.7.6.1. 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.**

- Open the Locations ViewObject and create an additional transient attribute named "LkpCtrCountryNameTransient". Set updateable to "always" and uncheck the queryable checkbox.
- In the Java Tab check the checkbox to create a LocationsViewRowImpl java class. Click OK to close the ViewObject editor
- Got to the newly created LocationsViewRowImpl class, and modify the getLkpCtrCountryNameTransient method as follows:

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

- 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.

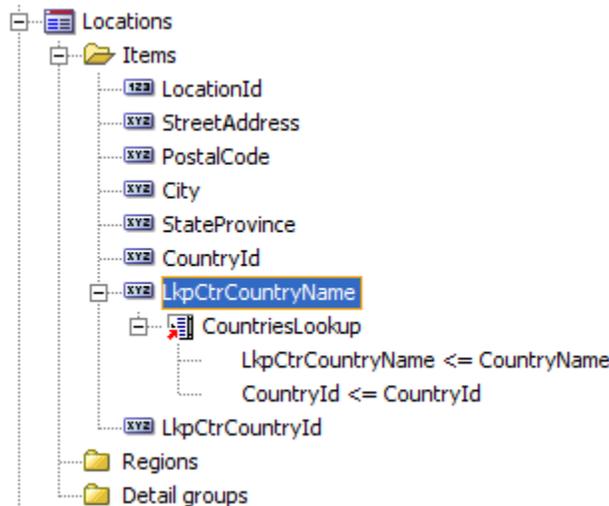
Now that we have extended the View Object, we need to make some changes to our application definition:

- On the Countries group set the **Group Usage** to 'List of Values Window'.
- 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.
- Set the **Display Type** of LkpCtrCountryName (or LkpCtrCountryNameTransient) to lov.
- Change the **Prompt in Form Layout** property of LkpCtrCountryName (or LkpCtrCountryNameTransient) to Country.
- Enable updates on LkpCtrCountryName (or LkpCtrCountryNameTransient) by setting the **Update allowed?** property to true.

Display Type *	lov
Display Settings	
Display in Form Layout? *	true
Display in Table Layout? *	true
Display in Table Overflo...	false
Prompt in Form Layout	Country
Prompt in Table Layout	
Width	#{bindings.
Height	#{bindings.
Maximum Length	30
Column Alignment	
Default Display Value	
Column Sortable?	<input checked="" type="checkbox"/>
Column Wrap?	<input type="checkbox"/>
Hint (Tooltip)	
Depends On Item	
Operations	
Update Allowed?	true

- 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.
- Select the LOV Countries (under LkpCtrCountryName) and add another return value by pressing the green plus (+) symbol.

- Change undefined `<= undefined` to `CountryId <= CountryId`



- Make `CountryId` and `LkpCtrCountryId` invisible by setting the **Display in Form/Table Layout?** properties to false.
- Generate and you get something like this.

* Location Id	<input type="text" value="1000"/>
Street Address	<input type="text" value="1297 Via Cola di Riei"/>
Postal Code	<input type="text" value="00989"/>
* City	<input type="text" value="Roma"/>
State Province	<input type="text"/>
Country	<input type="text" value="Italy"/>

6.7.7. 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:

- The user enters (part of) the lookup item value.
- The JHeadstart runtime checks how many records in the lookup match the value the user entered.
- 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.
- When zero or more than 1 record 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.

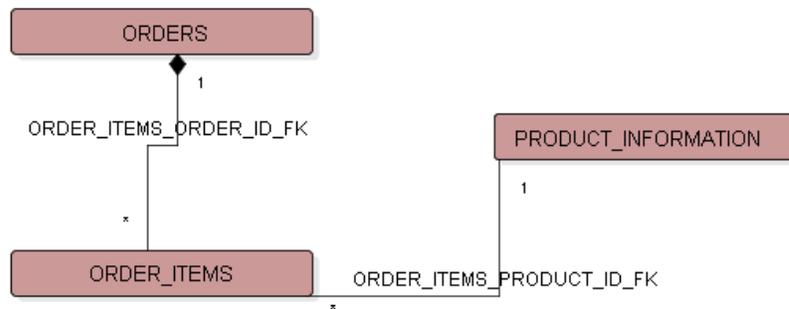
The next steps instruct you how to build this. The EMPLOYEES and DEPARTMENTS tables are used as example. The goal is to see the department name in the Employees page.

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 and run the application.
7. 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.
8. 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.

6.7.8. 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?**

Operations	
Multi-Row Insert allowed? *	<input checked="" type="checkbox"/>
Multi-Row Update allowed? *	<input checked="" type="checkbox"/>
Multi-Row Delete allowed? *	<input checked="" type="checkbox"/>

4. In the LOV group, set the property **Group Usage** to “List of Values Window” and check **Allow Multiple Selection in LOV?**.

Identification	
Name *	ProductsLov
Short Name	
Description	
Bound to Model Data Collection? *	<input checked="" type="checkbox"/>
Group Usage	List of Values Window
Group Region Access	
Allow Multiple Selection in LOV? *	<input checked="" type="checkbox"/>
Group Image / Icon	

5. Generate the application.



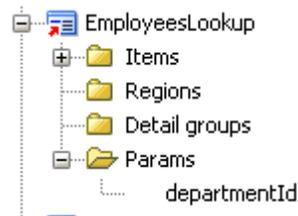
Suggestion: An alternative user interface for this situation is an Intersection Shuttle. See section 5.7.2. Creating Intersection Shuttles.

6.7.9. Passing Parameters to an LOV

You can pass parameters to an LOV to restrict the rows that are queried in the LOV based on the context in which the LOV was invoked. For example, an LOV on the `ManagerId` of a `Department`, might be restricted to see only managers that work in that same department.

Here are the steps to do this:

- You first need to define the parameter on the LOV group. Right-mouse-click on the LOV group, and choose **New Parameter...** Specify the parameter name, for example `departmentId`.



- In the underlying view object of the LOV group add a where clause that restricts the values based on a bind variable. For example:

```
department_id = :b_department_id
```

- In the LOV group, set the **Query Bind Parameters** property to pass the group task flow parameter as a value to the bind variable set up in the query of the view object. For example:

```
b_department_id=#{pageFlowScope.departmentId}
```

- In the LOV item that uses this LOV, add a parameter to pass the parameter value that should be used. The screen shot below illustrates this and uses the same example as above

6.7.10. Understanding How JHeadstart Runtime Implements List Of Values

6.7.10.1. Main ideas behind architecture

The implementation of the LOV component in JHeadstart is focused on **reusability** and **separation**. When you need an LOV on Employees, for example, you only have to create it once in your application and use it from anywhere. The LOV is also completely separated from the base page, meaning it does not interfere with the state of the base page.

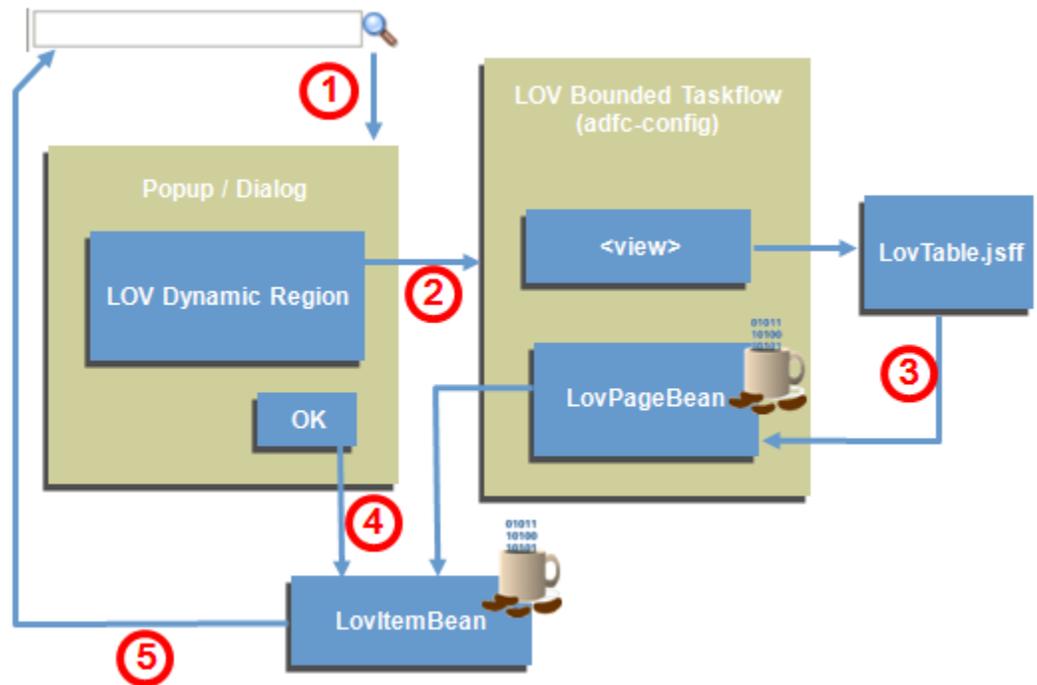
6.7.10.2. How it all fits together

The implementation of a JHeadstart LOV involves the following components:

- A generic popup component on the page. This popup window displays the LOV using a dynamic region. The same popup component is used for all LOV's on the page, which keeps the page size relatively small so page load time is not decreased by using many LOV's in a page.
- A SimpleDynamicRegionManager class with associated managed bean which controls the taskflow that is displayed inside the LOV dynamic region.
- The LOV group task flow with the LOV table page fragment that is displayed inside the popup.

- An LovPage bean that handles LOV selection event by passing the selected row data to the LovItemBean instance. The LovItemBean instance is passed as a task flow parameter to the LOV task flow.
- An LovItemBean instance for each LOV on the page. This bean contains information about the task flow that should be displayed in the popup, the parameters that should be passed in and how selected LOV values should be copied back to the base page.

How they all fit together in a normal LOV selection process is shown in the following diagram:



Step 1

When the user clicks the LOV icon, the popup is opened using the `lovIconClicked` method in the `LovItemBean`. Inside this method, we also “inform” the LOV Dynamic Region Manager which task flow should be displayed inside the popup and which parameters should be passed in. The attribute ‘contentDelivery’ of the popup is set to ‘lazyUncached’, meaning the client will only fetch the contents of the popup when it is opened (not before that time).

Step 2

When the popup is opened, the LOV dynamic region manager bean will switch to the taskflow of your LOV. By default, it will point to an empty taskflow, so the content of the popup is empty and resources are not wasted.

By switching the taskflow ID of the dynamic region manager, the LOV taskflow will be initiated. This taskflow will instantiate the `LovPageBean`, and will get the `LovItemBean` injected (which is a parameter for every LOV task flow). The purpose of the `LovPageBean` is to pass information about the selected row in the LOV to the `LovItemBean`. We will need this later on at step 5.

Step 3

When the taskflow is finished initializing, it will load the LOV fragment. The LOV always contains a result table. This table is component-bound to the LovPageBean, to enable this bean access to the selected row in the LOV table.

Now that everything is set up, the LOV is shown inside the popup and the user is now able to search and make a selection.

Step 4

When the user presses 'OK', the method 'handleLovSelection' in the LovItemBean is launched. This method is responsible for fetching the selected data in the LOV table, and copying those values back to the base page.

Step 5

In this final step, the LovItemBean will copy the values it retrieved from the selected row, to the fields and underlying value bindings on the base page.

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.

* Hire Date 

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

* Delivery Date 

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:

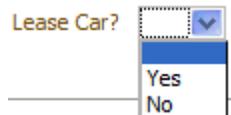
Lease Car?

In a table layout it will look like this:

Lease Car?
<input checked="" type="checkbox"/>
<input type="checkbox"/>
<input checked="" type="checkbox"/>



Attention: In a search region, IndLeaseCar will show as a dropdown list and not as a checkbox.



The reason is that we have three situations when searching:

1. We want to search for 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.

6.9.1. Generating a checkbox based on a Boolean attribute

If you have defined a transient view object attribute of type `java.lang.Boolean`, you do not need to set up a domain. In this case, you only need to set the **Display Type** to `checkbox`, and during the JSF Model Update phase, the setter method of this attribute will automatically be called with value `Boolean.TRUE` when the checkbox is checked, and with value `Boolean.FALSE` when the checkbox is not checked.

6.10. Generating a Button Item

JHeadstart provides extensive support for generation of (iconic) buttons, and executing various types of actions when the button is pressed. In this paragraph, the following topics are discussed:

- Positioning of buttons
- Appearance of buttons
- Executing a button action

6.10.1. Positioning of Buttons

The position of the button is determined by the **Display Type** of the item:

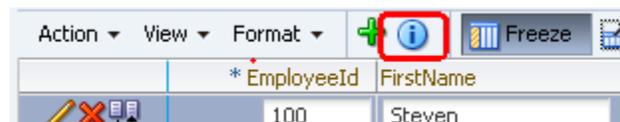
- `commandButton`: the button is located based on the position of the item relative to other items in the group or item region. In the screen shot below, the **Display At Right of Item** property is used to position the button item at the right of the `JobId` item.



- `toolbarButton`: if the **Display in Form Layout** property evaluates to true, the button is displayed in the form group toolbar.



If the **Display in Table Layout** property evaluates to true, the button is displayed in the table toolbar.



- `groupLinkButton`: the button is located based on the position of the item relative to other items in the group or item region.
- `groupLinkToolbarButton`: if the **Display in Form Layout** property evaluates to true, the button is displayed in the form group toolbar. If the **Display in Table Layout** property evaluates to true, the button is displayed in the table toolbar.

6.10.2. Appearance of Buttons

The appearance of the button is determined by the following properties:

- The **Prompt in Form Layout** or **Prompt in Table Layout** properties are used to generate a button with a text label.
- The **Icon** property is used to generate an iconic button. In this property you can specify a relative path from the html root directory to your image. A number of useful icon images can be found in the `/jheadstart/images` folder.

- The **Hint (Tooltip)** property can be used to generate a tool tip when the user hovers over the button with the mouse.



When both the prompt and an icon are specified, an iconic button with text is generated as shown below.



6.10.3. Executing a Button Action

There are a number of methods to associate an action with a button that should be executed when the button is pressed:

- A popup window is launched when a region container with **Display Type** `Modal Popup Window` or `Modeless Popup Window` has the **Depends on Item(s)** property specified referring to a button item. See chapter 5, section 5.10.4 [Generating Content in a Popup Window](#) for more information.
- When the **Display Type** of the button item is set to `groupLinkButton` or `toolbarGroupLinkButton`, the called group will be displayed, either in page, or in a modal or modeless popup window, or in a new dynamic tab. See section [Navigating Context-Sensitive to Another Group Taskflow \(Deep Linking\)](#) for more information.
- When the **Method Call** property is set, the business method specified in this property is executed.
- When the **Action** property is set the action method specified in the property will be executed, or when a literal value is specified navigation might take place based on this value.
- When the **ActionListener** property is set the action listener method specified in the property will be executed.

The last three methods are discussed in more detail below.



Attention: By default, the button is generated without the `immediate="true"` property. This means that the JSF phases `Process Validations` and `Model Update` are executed before the button action is performed in the `Invoke Application` phase. This is useful when the user enters values on the screen, and then presses a button that should perform an action where these values are needed. The drawback is that if you have invalid data on your page, an error message will be displayed and the button action will not be performed.

If you do not want validations to fire, you can use the expert-mode property **Additional Properties** and specify `immediate"true"` as the value. Do remember that when `immediate` is set to `true`, any values just entered on the screen are not propagated to the ADF binding layer (and subsequently to ADF Business Components) because the JSF `Model Update` phase is skipped in this case.

6.10.3.1. Using the Method Call Property

To use this property, you need to create a custom method on the application module and publish this method on the client interface. Once, you have done this, the method appears in the **Method Call** dropdown list.

As an example, we will use a method to increase the salary of an employee to illustrate the steps to make this work.

- Go to the Model project, and find the application module that is specified as data control in your JHeadstart service definition.
- Go to the Application Module Class. 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
    ViewObject 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.

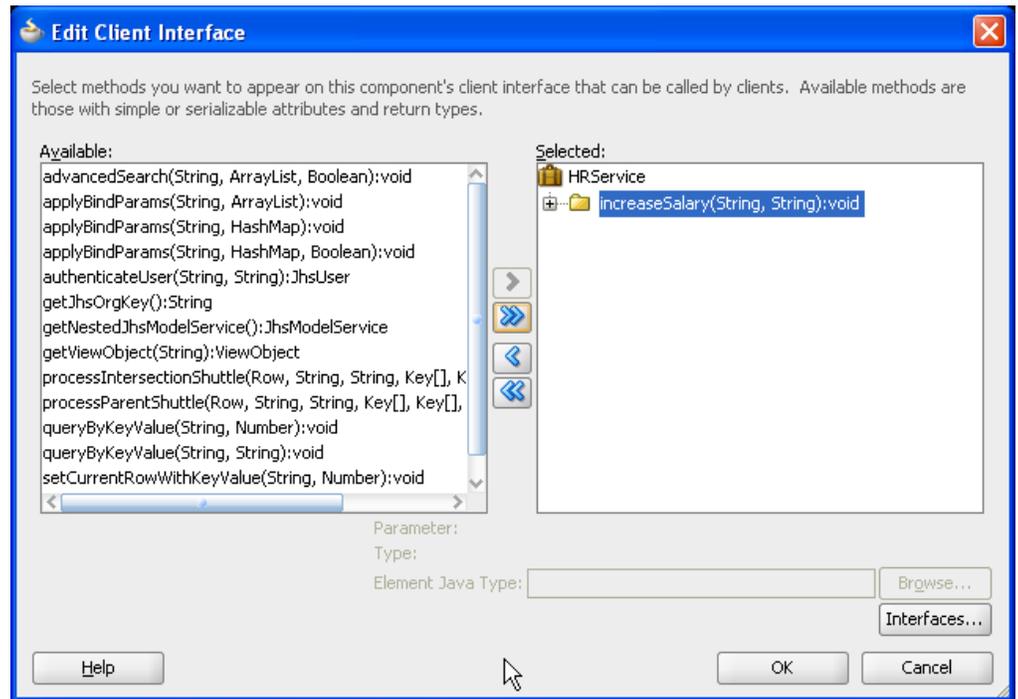


Attention: This code assumes that you created a `ViewRowImpl` class for the `EmployeesView`. You can generate this class by going to the `EmployeesView` editor, selecting the Java category and checking the `Generate View Row Class` checkbox.

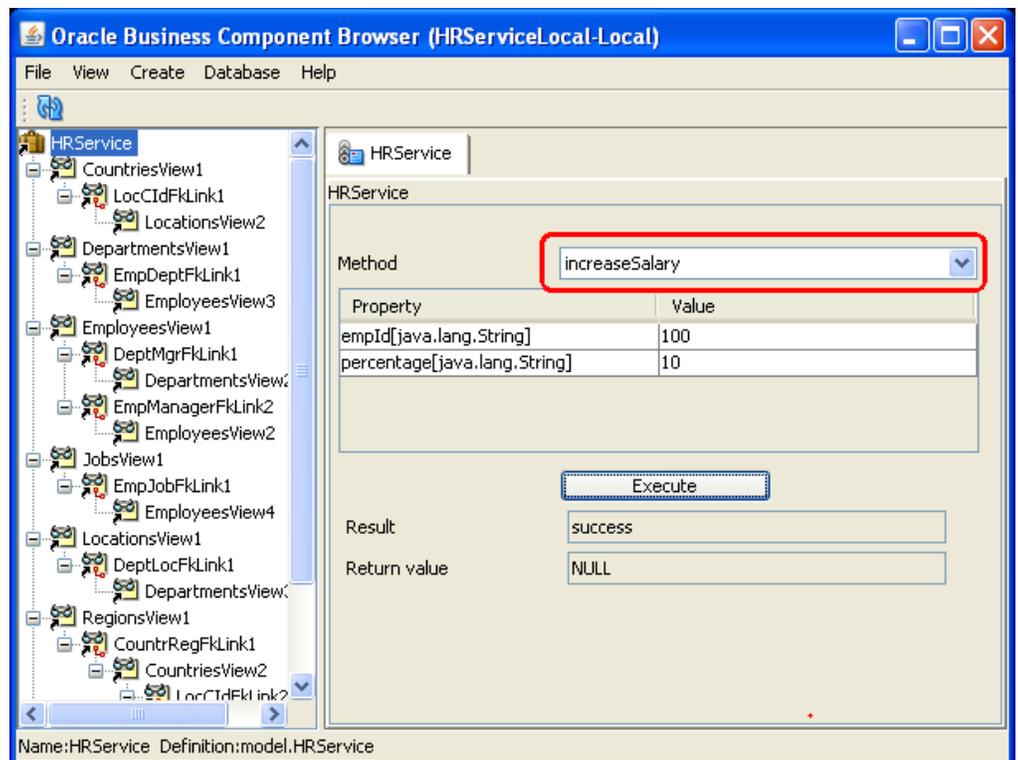


Attention: The `Number` class referenced is `oracle.jbo.domain.Number`, not `java.lang.Number`.

- Publish this method on the application module client interface. In the Application Module Editor, go to the Java category, and edit the Client Interface by clicking the pencil icon and shuttle the new `increaseSalary` method to the right.



- Optionally, you can run the Application module tester to test the method by double clicking the application module node.



You can now define a new unbound item named `IncreaseSalary` with **Display Type** `commandButton` within the JHeadstart Application Definition Editor and set the **Method Call** property to call the `increaseSalary` method.

Action Listener	
Method Call	increaseSalary
Icon	increaseSalary

To pass in the method parameters, we add two parameters to the item. In the parameter **Name** property you will get a dropdown list with the method arguments.



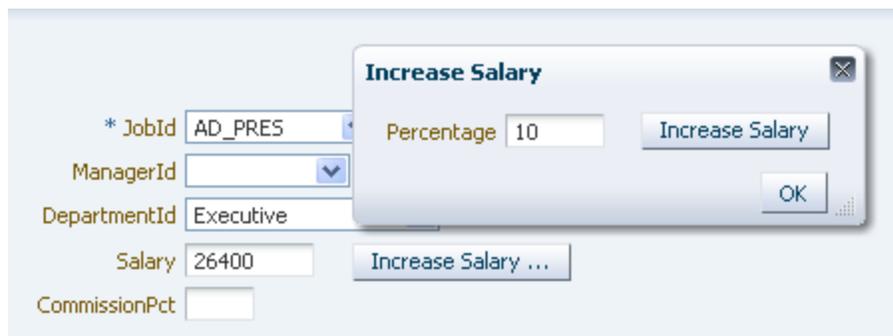
For `empId`, we need to pass in the value of the current employee. We use the attribute binding generated for the `EmployeeId` item for this. JHeadstart-generated attribute bindings follow the naming convention `[group (Short) Name] [itemName]`, so since the group name is `Employees` and no **Short Name** is defined, the name of the attribute binding is `EmployeesEmployeeId`. The complete expression to get the value of this attribute binding is:

```
#{bindings.EmployeesEmployeeId.inputValue}
```

For the `percentage` parameter you can fill in a fixed value like 10, or, if you want to make this user-enterable, you can define another unbound item in the group where the user can enter a percentage. If you create an unbound item named `Percentage`, the EL expression to pass the value is similar to getting the employee id:

```
#{bindings.EmployeesPercentage.inputValue}
```

You could even show a popup window using the technique described in chapter 5, section 5.10.4 *Generating Content in a Popup Window*, where the user can enter the percentage, and click the `increaseSalary` button, as shown below.



The steps to do implement this are:

- Set the **Layout Style** of the Regions region container to `Modal Popup Window`.
- Add an item region inside the region container with the `Percentage` item and the `IncreaseSalary` button item.
- Create an additional button item `LaunchIncreaseSalaryPopup` and set the **Depends on Item(s)** property of the region container to this item.
- Set the **Depends on Item(s)** property of `Salary` to the `IncreaseSalary` button item to immediately see the updated salary after the button has been pressed.

6.10.3.2. Using the Action Property

The use of the **Action** property is identical to (and generated as) action property of a JSF command button: you can specify a literal string which is used for navigation, or you can specify a no-argument managed bean method that returns a string value that can be used for navigation.

You can easily generate a custom managed bean that contains the method you want to call in the group task flow. See chapter 13, section 12.5.5 Adding Custom Managed beans for more information.

6.10.3.3. Using the Action Listener Property

The use of the **Action Listener** property is identical to (and generated as) action listener property of a JSF command button: you can specify a void managed bean method that takes the JSF `ActionEvent` as the single argument.

You can easily generate a custom managed bean that contains the method you want to call in the group taskflow. See chapter 13, section 12.5.5 Adding Custom Managed beans for more information.

6.10.4. Calling a PL/SQL Procedure or Function From a Button

To call PL/SQL logic in the database, you first create an application module method that calls the database procedure or function, and then you can use the **Method Call** property to call the application module method as described in the previous section.

The JHeadstart runtime library contains a convenience class, `oracle.jheadstart.util.DatabaseProcedure`, which makes it very easy to call PL/SQL in an intuitive way. You basically copy the PL/SQL specification of the procedure or function, and then just call it. The functionality of this class is best explained through some examples

6.10.4.1. Simple PL/SQL Function Call Example

To call a database function `hello_world` and get the result of the call in a `String` variable, one only needs to use the following lines of code:

```
String name = "John";
DatabaseProcedure helloWorldProc =
DatabaseProcedure.define("function hello_world(p_name in varchar2)
return varchar2");
String result = (String) helloWorldProc.call(getDBConnection(),
name).getReturnValue();
```

6.10.4.2. Simple PL/SQL Procedure Call Example

To call a database procedure with several output parameters, do the following.

```
Integer empId = 100;
DatabaseProcedure getEmpDetailsProc =
DatabaseProcedure.define("procedure get_employee_details
( p_id in number
, p_name out varchar2
, p_address out varchar2
, p_age out number)");
DatabaseProcedure.Result result =
getEmpDetailsProc.call(getDBConnection(), empId);
```

```
String name = (String) result.getOutputValue("p_name");
String address = (String) result.getOutputValue("p_address");
Number age = (Number) result.getOutputValue("p_age");
```

6.10.4.3. Calling a function/procedure inside a package

To call a function or a procedure that resides inside a package, just prefix the name of the function/procedure with the package name.

```
DatabaseProcedure helloWorldProc =
DatabaseProcedure.define("function my_package.hello_world(p_name
in varchar2) return varchar2");
```

6.10.4.4. Recommended Use

A DatabaseProcedure object can be reused multiple times. It is therefore a best practice to store the DatabaseProcedure object in a static variable (as a constant). Then you create an application module method that make the actual database call. The hello world example shown earlier looks like this:

```
private static final DatabaseProcedure HELLO_WORLD =
    DatabaseProcedure.define("function hello_world(p_name in varchar2) return varchar2");

public String helloWorld(String name)
{
    return (String) HELLO_WORLD.call(getDBTransaction(), name).getReturnValue();
}
```

Publish this method on the application module client interface, test it by running the application module tester, and if it works correctly, you can use the **Method Call** property in the JHeadstart Application Definition Editor and select the method from the dropdown list.



Attention: The PL/SQL specification you use is parsed by the Java code only once (using regular expressions). No database call is made until you actually use the `call()` method. At that point it simply uses ONE ordinary JDBC call. So no extra calls have to be made and the behavior is the same as ordinary JDBC. Therefore, it is very unlikely that you will encounter any measurable performance penalty at all.



Tip: You can directly select and copy the procedure or function specification in your favorite PL/SQL editor, and then paste it inside the brackets of the `define` method argument in JDeveloper. JDeveloper will then nicely add all the new line feeds you also used in the PL/SQL editor, which makes the code much more readable when using procedures or functions with many IN or OUT parameters.

6.11. 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, ORDSYS.ORDAUDIO and BLOB.



Attention: The abovementioned types that start with ORDSYS are object types defined in the Oracle *interMedia* 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 *interMedia* section of the Oracle Technology Network (<http://otn.oracle.com/products/intermedia>).

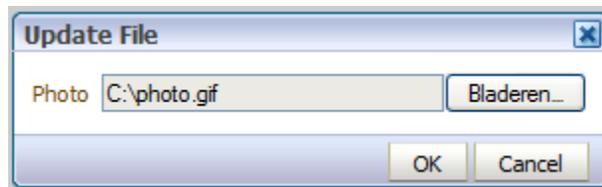
Example of generating a file upload field in the HR sample schema for uploading photos of employees:

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:

Photo myphoto.gif Update...

4. When you click the Update button, you will get a popup that looks like this:



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

5. If instead, you want that field to display the photo, you can change the **Display Type** of the item to *image*.
6. 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!):

Photo



If the file you uploaded was an audio file (this is possible with the BLOB, 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, check the Configuration appendix of the Web User Interface Developers Guide for ADF, at http://download.oracle.com/docs/cd/E17904_01/web.1111/b31973/ap_config.htm#sthref2668

6.11.1. 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?	<input checked="" type="checkbox"/>
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:

Photo myphoto.gif Update...



6.11.2. 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`, `ORDAUDIO` or `BLOB`).

Here are some properties that you might want to show:

Property	SQL name	ADF BC method	Java type
File Size (in bytes)	<code>ContentLength</code>	<code>getContentLength()</code>	<code>int</code>
File Mime Type	<code>MimeType</code>	<code>getMimeType()</code>	<code>String</code>
(Original) File Name	<code>source.srcName</code>	<code>getSourceName()</code>	<code>String</code>
File Upload Time	<code>source.updateTime</code>	<code>getUpdateTime()</code>	<code>Timestamp</code>

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.
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().getLength());
        }
        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.

Download File [Configuration.xml](#)

6.12. Generating a Graph

The number of possible types of graphs with the ADF Data Visualization components is huge, as well as the customization possibilities and wizard options.



Fusion Developers Guide, chapter 26 “Creating Databound ADF Data Visualization Components”. Includes instructions on defining graphs, gauges, gantt charts, maps and pivot tables
http://download.oracle.com/docs/cd/E17904_01/web.1111/b31974/graphs_charts.htm#CHDIBDEF

We do not aim to copy (a mere subset of) these features in JHeadstart, but let you use the best of both worlds: the very rich wizards on ADF Data Visualization components, plus the flexibility to integrate the graphs into your JHeadstart application.

EmployeeId	FirstName	LastName	Salary
113	Luis	Popp	6900
111	Ismael	Sciarra	7700
112	Jose Manuel	Urman	7800
110	John	Chen	8200
109	Daniel	Faviet	9000
			7920

In the screen shot above, we create an item with **Display Type** graph, and placed in the overflow right area. As you can see the item generated is just used as a place holder for the actual graph that you need to add manually using drag and drop, as shown below.

EmployeeId	FirstName	LastName	Salary
109	Daniel	Faviet	9000
110	John	Chen	8200
111	Ismael	Sciarra	7700
112	Jose Manuel	Urman	7800
113	Luis	Popp	6900
			7920

After the graph is displayed correctly, you can create a custom template for the graph item to preserve your graph customizations when regenerating your application. For detailed steps on how to do this, see the JHeadstart Tutorial at <http://download.oracle.com/consulting/jhstutorial1111.pdf>

6.13. 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 Item\(s\) 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](#).

6.13.1. Using the Depends On Item(s) 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.

6.13.1.1. 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 CommissionPct Item to Depend on the JobId Item** by setting its **Depends on Item** property to JobId.

JobId	Column wrap? *	<input type="checkbox"/>
Salary	Hint (Tooltip)	
CommissionPct	Depends On Item(s)	JobId
ManagerId	Clear/Refresh Value? *	<input type="checkbox"/>
DepartmentId		

- 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'`). If you change the `JobId` of Hartstein to `AC_MGR` using the dropdown list, you will see that the `CommissionPct` item is enabled immediately.

6.13.1.2. 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.

JobId	Column wrap? *	<input type="checkbox"/>
Salary	Hint (Tooltip)	
CommissionPct	Depends On Item(s)	JobId, DepartmentId
ManagerId	Clear/Refresh Value? *	<input type="checkbox"/>
DepartmentId		

- 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.

6.13.2. 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 area 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.

6.13.3. 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!

6.14. Navigating Context-Sensitive to a Group Task Flow (Deep Linking)

JHeadstart allows you to 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.

In this example, we will use the JHeadstart deep linking support to generate the `JobId` in the `Employees` group table and form as a hyperlink that navigates to the job edit page, querying the proper job row.

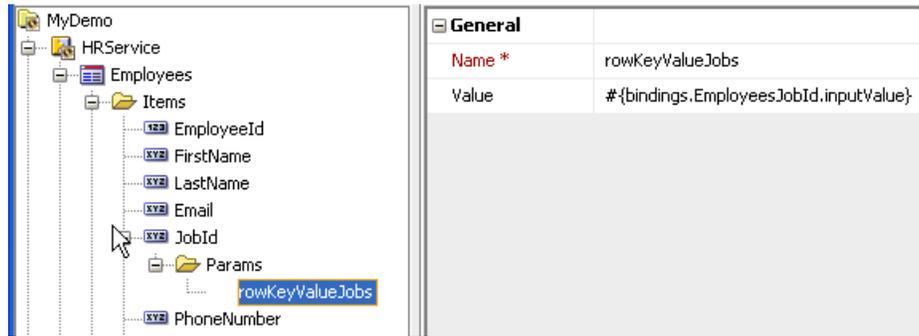
To get the deep linking to work, follow these steps:

1. Open the JHeadstart Application Definition Editor, and navigate to the item that you want to display as a hyperlink (in this case: the item `JobId` in the `Employees` group).
2. Set the **Display Type** to one of the following:
 - `groupLink`: generates a hyperlink to perform the group navigation
 - `groupLinkButton`: generates a text button, or iconic button (if you specify the **Icon** property) to perform the group navigation
 - `groupLinkToolBarButton`: generates a text button, or iconic button (if you specify the **Icon** property) in the group toolbar to perform the group navigation

In this example we will use **Display Type** `groupLink`.



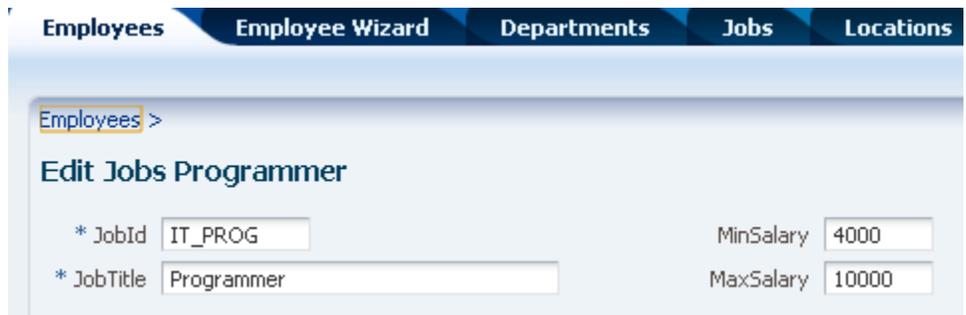
3. Right click the item, and add a Parameter.
4. Click in the **Name** property of the parameter and choose `rowKeyValueJobs` from the dropdown list. The `rowKeyValue[groupName]` parameter is one of the standard parameters generated for each task flow to enable deep linking. For more information on these standard task flow parameters, see chapter 12, section 12.5.1 Understanding Generated task Flow Structure.
5. Set the **Value** of the parameter to `{bindings.BaseGroupNameItemName.inputValue}`, which is for the example `{bindings.EmployeesJobId.inputValue}`.



- That's it! Save your application definition and generate the application. The resulting Employees table will now look like this:

	FirstName	LastName	Email
▶	Steven	King	SKING
▶	Neena	Kochhar	NKOCHHAR
▶	Lex	De Haan	LHAAN
▶	Alexander	Hunold	AHUNOLD
▶	Bruce	Ernst	BERNST
▶	David	Austin	DAUSTIN
▶	Valli	Pataballa	VPATABAL
▶	Diana	Lorentz	DLORENTZ

When you click the link, you will go to the Jobs page with the Job you clicked on as the current row:



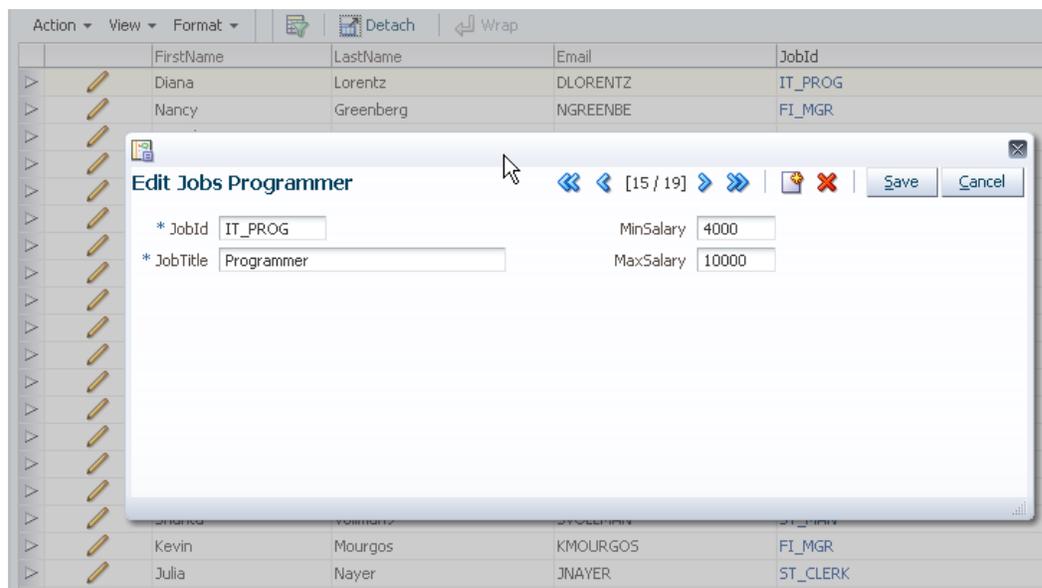
Note that if you want to generate a button rather than a link to navigate to the other group, you can easily make a custom template based on the default GROUP_LINK template and replace the af:commandLink tag with the af:commandButton tag.

6.14.1. Opening the Linked Group in Popup Window

You can use the property **Show Linked Group In** to determine how the target group is made visible. The default is "In Page" which causes the above in-page navigation where the deeplink page of the target group replaces the page with the group link. You can also show the linked group in a modal or modeless popup window, or in a new dynamic tab (see next section for this last option). You can then use the items **Width** and **Height** properties to set the size of the popup window.

Display Type *	groupLink
Link Group Name	HRService.Jobs
Show Linked Group In	Modal Popup Window
Display Settings	In Page
Display in Form Layout? *	Modal Popup Window
Display in Table Layout? *	Modeless Popup Window
Display in Table Overflow Area? *	New Dynamic Tab
XYZ Display at Right of Item	
Display Summary Type in Table	
Prompt in Form Layout	#{\$HINTS\$.label}
Prompt in Table Layout	
Short prompt	
Width	400
Height	300

If you generate with these settings, clicking the group link launches a popup window showing the deep link form page of the target group task flow.



6.14.2. Using Taskflow Parameters to Customize Appearance of Linked Group

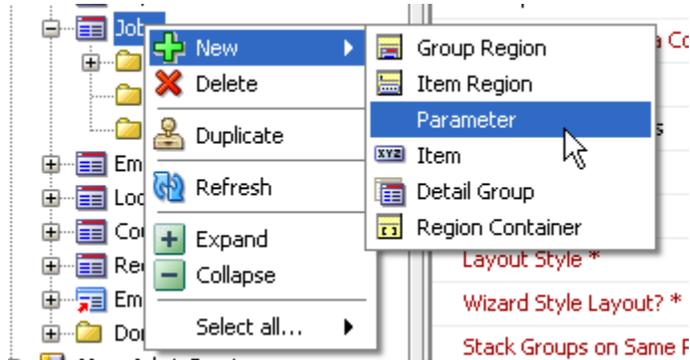
In addition to the task flow parameters that can be used to set the current row in the target group, you can use parameters to customize how the pages of the target group appear. Each group task flow generated by JHeadstart has the following standard boolean parameters to customize the appearance:

- hideSaveButton
- hideCancelButton
- hideSearchArea
- hideFormBrowseButtons

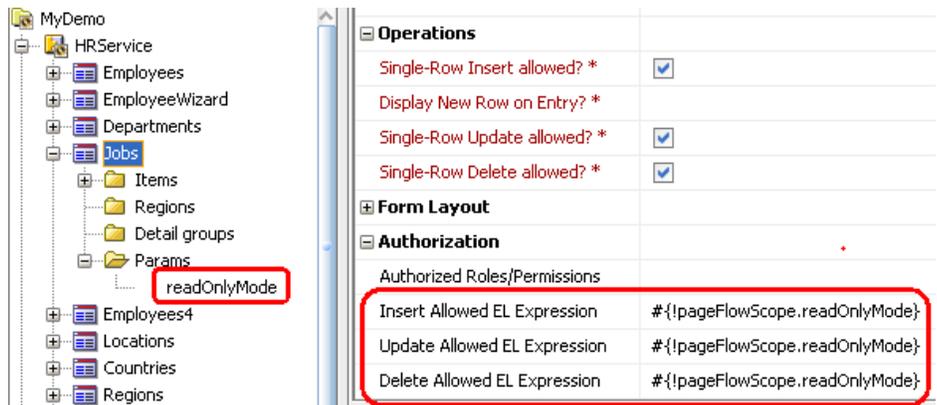
Note that these parameters are defined in the bounded task flow template. You can add additional parameters to all groups by making a custom JHeadstart Velocity template to generate the bounded task flow template.

You can also add group-specific custom task flow parameters. For example, we can add a task flow parameter to show the Jobs deep link page in readOnly mode, as follows:

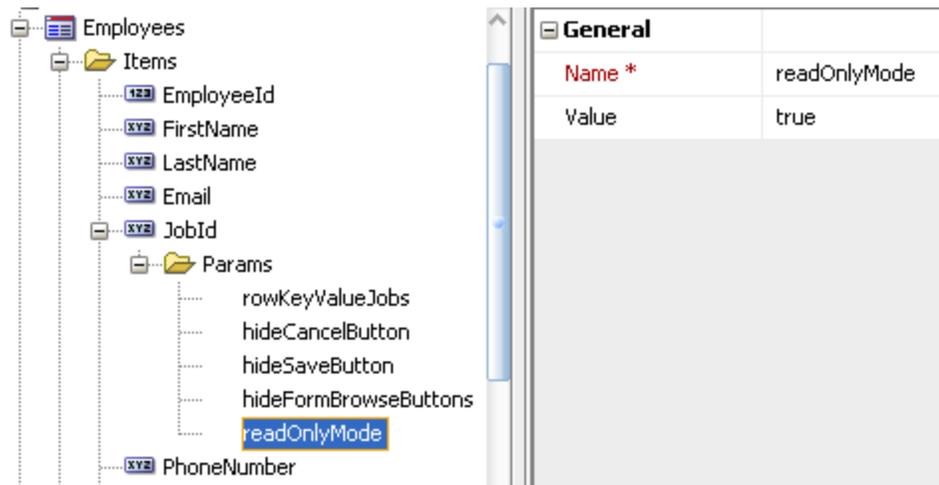
- Right-mouse-click on the Jobs group and choose New -> Parameter.



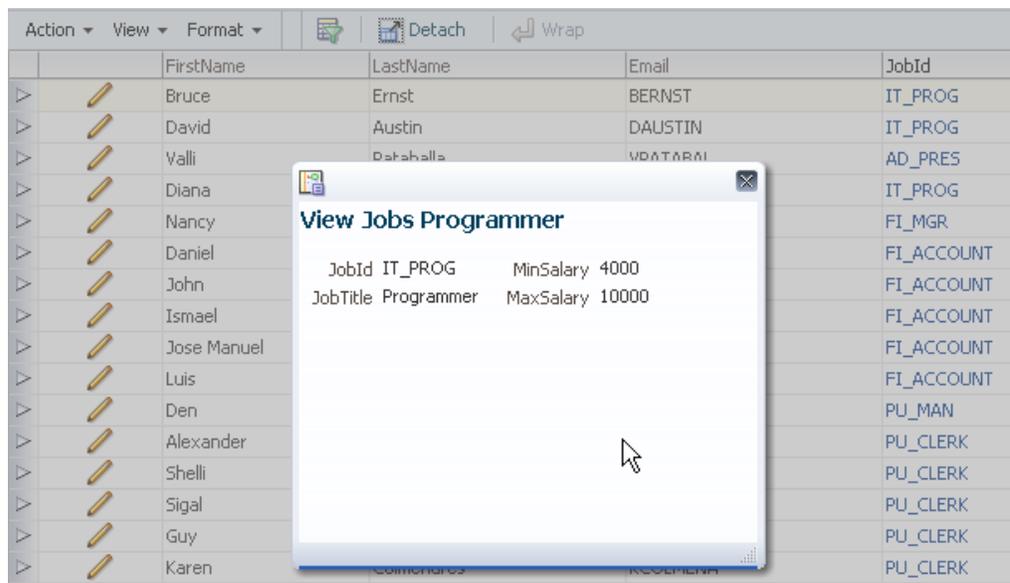
- Set the name of the parameter to “readOnlyMode”
- Set the **Insert Allowed EL Expression**, **Update Allowed EL Expression** and **Delete Allowed EL Expression** to the reverse value of the readOnlyMode task flow parameter, as shown below.



Now that we have added this custom task flow parameter to the Jobs group, we can add additional parameters to the group link item in the Employees group, to show the Jobs group in read-only mode without Cancel, Save and Form Browse buttons:



When we generate with these settings and click the JobId hyperlink, the popup will look like this:



6.14.3. Opening the Linked Group in a New Dynamic Tab

You can enable dynamic tabs at the application level. When you do this you create a multi-tasking user interface within the ADF-JHeadstart application where the end user can open multiple tabs with independent tasks (and independent transaction boundaries), just like browser tabs. See chapter 9 "Creating Menu Structures" for more information on enabling dynamic tabs in your application.

If dynamic tabs are enabled at the application level, you can choose the value **New Dynamic Tab** in the **Show Linked Group In** property. A typical use case for this option is the pattern where you have a search page for a key domain object in your application, like employees in an human resources application, and then you can select one or more instances to work with, that open in separate tabs.



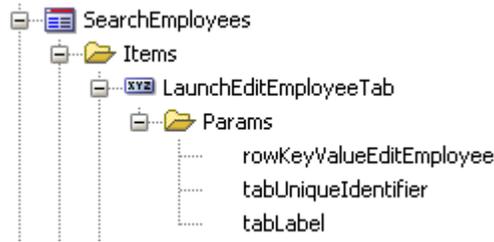
Here are the steps to implement this for employees, as shown above:

- Set the application-level property **Page Template** to `/common/pageTemplates/JhsDynamicTabsPageTemplate.jsf`
- Set the application-level property **Data Control Scope** to `Isolated`.
- Create two groups, `SearchEmployees` and `EditEmployee` that are based on the `EmployeesView1` view object usage.
- `SearchEmployees` group: Set the **Layout Style** to `table`, the **tabName** to `Search Employees` and make the table read-only by unchecking the **Multi-Row Insert/Update/Delete Allowed** properties.
- Set the **Layout Style** for `EditEmployee` group to `form` and uncheck the checkbox **Add Menu Entry for this Group**
- Create a new unbound item `LaunchEditEmployeeTab` in the `SearchEmployees` group, make this the first item in the group, and set the properties as shown below.

General	
Bound to Model Attribute? *	<input type="checkbox"/>
Name *	LaunchEditEmployeeTab
Short Name	
Value	
Java Type *	String
Display Type *	groupLinkButton
Link Group Name	HRService.EditEmployee
Show Linked Group In	New Dynamic Tab
Icon	/jheadstart/images/editRow.png
Display Settings	
Display in Form Layout? *	true
Display in Table Layout? *	true
Display in Table Overflow Area? *	false

- Add the `rowKeyValueEditEmployee` parameter, with value `#{bindings.SearchEmployeesEmployeeId.inputValue}`

- Add the `tabUniqueIdentifier` parameter, with value `#{bindings.SearchEmployeesEmployeeId.inputValue}`. By specifying this parameter, clicking on the edit icon for an employee that already has an edit tab open will select this existing tab instead of opening a new tab. Note that if you do not specify this parameter, JHeadstart uses the first parameter value as tab unique identifier which in this case would have been OK.
- Add the `tabLabel` parameter with value `#{nls['EDIT_TITLE_EDITEMPLOYEE:#{bindings.SearchEmployeesLastName.inputValue}']}` }



If the `tabLabel` parameter is not specified, the `tabName` property of the group is used.

6.14.4. Deep Linking from an External Source

You can also start the application and directly open a specific group task flow and specific row within this task flow from an external source, like a link in another application or from e-mail. To do this, you need to specify a request parameter named `jhsTaskFlowName` in the URL with a value that matches a group name as defined in the JHeadstart Application Definition Editor. Any standard or custom task flow parameters defined for this group task flow can also be specified as request parameter in the URL. For example the following URL will start the application with the `Employees` taskflow, displaying employee with `EmployeeId` 110:

```
http://127.0.0.1:7101/MyJhsApp/UIShell?jhsTaskFlowName=Employees
&rowKeyValueEmployees=110
```

If security is enabled, you will first get a login page, and then you will be redirected to the specific task flow page.

6.15. Generating Embedded Oracle Forms

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:

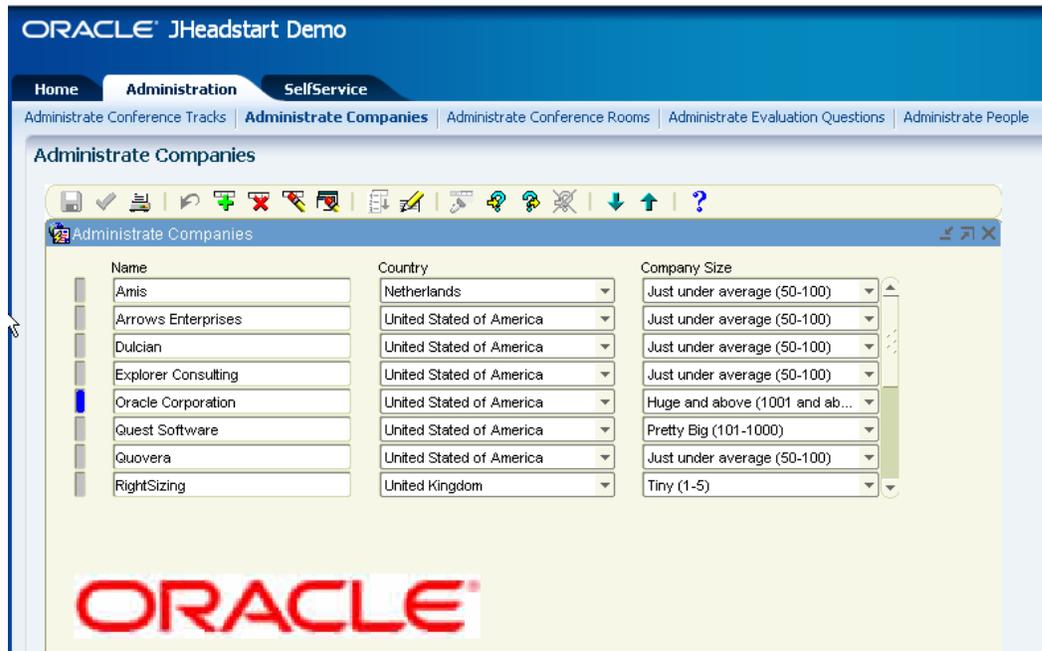
- Install OraFormsFaces in JDeveloper, by following the instructions in the OraFormsFaces Developer’s Guide.
- Add the OraFormsFaces environment entries in the web.xml of your project, as documented in the OraFormsFaces Developer’s Guide.
- Create a new top-level group in the application definition editor. 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.

The screenshot shows the JDeveloper IDE. On the left, the application definition editor displays a tree structure for an application named 'OCM'. Under the 'Items' folder, an item named 'oraForm' is selected. On the right, the Properties window is open, showing the configuration for the selected 'oraForm' item. The 'General' tab is active, and the 'Display Type' property is highlighted with a red box, showing its value as 'oraFormsFaces'. Other visible properties include 'Name' (oraForm), 'Value' (OCM0020), and 'Java Type' (java.lang.String).

General	
Bound to Model Attribute? *	<input type="checkbox"/>
Name *	oraForm
Short Name	
Value	OCM0020
Java Type *	java.lang.String
Display Type *	oraFormsFaces
Display Settings	
Display in Form Layout? *	true

- 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!



6.15.1. Deep Linking to an Oracle Form

You can use the normal deep linking facilities in JHeadstart to navigate to an embedded Oracle Form and query the context in this form. At the Oracle Forms side, the form must contain a global or forms parameter that will receive the context information required to query a specific row. And the Oracle Form needs to have a WHEN-NEW-FORM-INSTANCE or PRE_FORM trigger that will execute the appropriate logic based on the value of the form parameter or global.

In the following example, we will deep link from a JHeadstart-generated table page that shows person information to an Oracle Form that can be used to maintain the person information. The Oracle Forms contains a parameter named PEOPLE_ID.

Here are the steps to follow:

- In the group that contain the oraFormsFaces item, add a parameter named PEOPLE_ID

- In the oraFormsFaces item, add a parameter also named PEOPLE_ID, and set the value to the EL expression that references the group parameter that will be generated as a taskflow parameter: `${pageFlowScope.PEOPLE_ID}`

General	
Name *	PEOPLE_ID
Value	<code>#{pageFlowScope.PEOPLE_ID}</code>

- In the group that should generate a hyperlink to call the embedded form, create an item with **Display Type** "groupLink" and specify the **Link Group Name** and **Show Linked Group In** properties.

General	
Bound to Model Attribute? *	<input checked="" type="checkbox"/>
Name *	LastName
Short Name	
Attribute Name *	LastName
Value	
Java Type *	java.lang.String
Display Type *	groupLink
Link Group Name	SelfService.OCM0140
Show Linked Group In	In Page
Display Settings	
Display in Form Layout? *	false
Display in Table Layout? *	true

- Add a parameter named PEOPLE_ID to this item and set the value to the EL expression that references the primary key value.

General	
Name *	PEOPLE_ID
Value	<code>#{row.Id}</code>

- That's all. Generate and run the application!

The calling page with the hyperlink looks like the picture below.

The screenshot shows a web application titled "Administrate People". It has a "Filter By" dropdown set to "First Name". Below the filter are buttons for "Action", "View", "Format", and "Freeze". A table lists several people:

	First Name	Last Name
	Ken	Atkinsqqq
	Frank	Brink
	Bradley	Brown
	David	Brown
	Steven	Davelaar
	Paul	Dorsey

And clicking the link will query the correct row in the Oracle Form as shown below.

The screenshot shows the "Conference Attendance Management" form. At the top, it says "Administrate People (Davelaar)" and "Conference Attendance Management". Below that is a toolbar with various icons. The main form area is titled "Conference Attendees" and contains the following fields:

- First Name: Steven
- Last Name: Davelaar (highlighted with a red box)
- Email: steven.davelaar@oracle.com

There are two tabs: "Personal Details" (selected) and "Company". Under "Personal Details", there are:

- Country: Netherlands
- Odtug Member
- Biography: He created Headstart, he created JHeadstart and he will create much more. The master of generation is now the Knight of Reuse.
- Birthdate: 21-SEP-1963

At the bottom, there is a "Presenter Of" section with a table:

Day	Slot	Title
Tuesday 18th June	3.15 -4.15 PM	Oracle JHeadstart - Building E-Business Applications
Wednesday 19th June	2.00 -3.00 PM	CDM RuleFrame: the framework for analysing, design



Generating Query Behaviors

This chapter discusses how to add query behaviors to a web page. Topics discussed include:

- Using ADF Model search or JHeadstart custom search
- Using auto-query
- Using Query Bind Parameters
- Quick Search
- Advanced Search
- Forcing a Requery

7.1. 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 choices and preparations.

7.1.1. Choosing a Technology: ADF Model or Custom JHeadstart

For both quick and advanced search you can choose between:

1. **ADF Model approach.** With this approach you define your quick and/or advanced search in the View Object itself using View Criteria. The page has a single component (`<af:query>` or `<af:quickQuery>`) that renders a complete search area.
2. **Custom JHeadstart approach.** Here you don't specify anything in the View Object, but use the meta-information in JHeadstart to generate a custom quick or advanced search area.

The choice of this technology also has to do with the choice for the technology behind List of Value components (see Chapter 6 and the compatibility matrix there).

In general, we recommend to use the ADF Model approach, unless you have requirements that cannot be implemented using the ADF Model search. Such requirements include:

- Specific layout requirements for the search area, for example, organizing the search items in groups with headers, or multiple tabs.
- Read-only descriptor items that should be populated by an LOV for a search item
- Usage of JHeadstart List of Values in the search area.

7.1.2. Creating a Model Based Search

To create a model based search, you can optionally specify View Criteria in Business Components. See the link below how to perform this.



Reference: Fusion Developers Guide for ADF, section 5.11 “Working with Named View Criteria”.

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/bcquerying.htm#BCGIFHHF

To specify that you want to use ADF Model based advanced or quick search, go to the JHeadstart Application Definition Editor, and click the applicable group. Go to search settings and choose ‘model-samePage’ for **Advanced Search?** and ‘model’ for **Quick Search?**

Search Settings	
Advanced Search?	model-samePage
Advanced Search View criteria	
Quick Search?	model
Quick Search View criteria	

Even if you have not specified any View Criteria, you can now generate your application and the group will contain a default quick and advanced search, where the user can search on all attributes. That is, all attributes that have ‘Queryable?’ checked in the View Object Attribute editor.

You can also select a View Criteria that you have made in your View Object. Suppose we have a ViewCriteria that selects employees with a salary above 5000 and called it ‘HighSalaryEmployees’. We could now select that for our View Criteria and it will be used as the **default** View Criteria:

Search Settings	
Advanced Search?	model-samePage
Advanced Search View criteria	HighSalaryEmployees
Quick Search?	model
Quick Search View criteria	
Single or Default Search Item	HighSalaryEmployees
Auto Query? *	<input type="checkbox"/>

After generating, the quick search will look like this:



And when you press the ‘Advanced Search’ button, it will look like this:

Notice how you can only set the attributes that appear in the View Criteria that was specified (HighSalaryEmployees). If you would not set any ViewCriteria, you would be able to set any column in the View Object.



Reference: Web User Interface Developer's Guide for ADF, chapter 14 “Using Query Components”. Explains in more detail how the `<af:query>` and `<af:quickQuery>` components work.
http://docs.oracle.com/cd/E24382_01/web.1112/e16181/af_query.htm#BABDIHB
[A](#)

7.1.3. Creating a Custom JHeadstart Search

When creating a custom JHeadstart search, you typically leave the ADF Business Components untouched and do not define any View Criteria.

Instead, you need to review each group in the JHeadstart Application Definition and identify items that should not logically be queryable.

If requested to generate search functionality, the JHeadstart Application Generator needs to know what the queryable items are. You can set the **Include in Quick/Advanced Search** properties for each item in a group. Both properties default to true.

1. Select an item in the Application Definition Editor.
2. Check or uncheck the **Include in Quick/Advanced Search** properties.

Query Settings	
Include in Quick Search?	<input checked="" type="checkbox"/>
Include in Advanced Search?	<input checked="" type="checkbox"/>

7.1.4. Using JHeadstart 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. Set the **Quick Search?** property to 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 queryable 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'`.

7.1.5. Using JHeadstart 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.



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 item 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.

7.1.6. Using a JHeadstart 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

- Go to the 'Advanced Search' region in the generated application. You will see something like this:

- JHeadstart has generated a dropdown list with applicable query operators for this field.

7.1.7. Using Query Bind Variables in JHeadstart Quick or Advanced Search

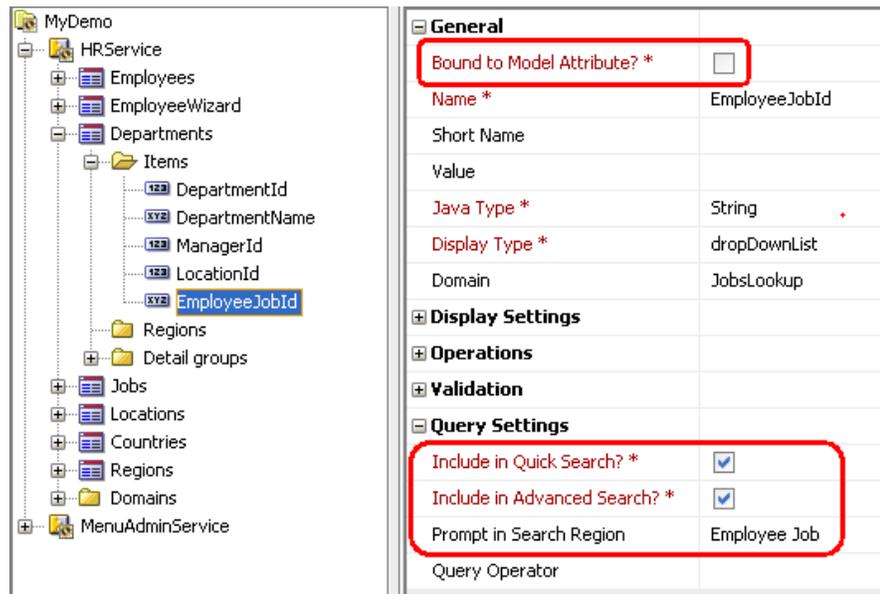
Based on the quick or advanced search items that are set by the user, a query WHERE clause is appended dynamically to the query (see section [Search Support in ADF BC Application Module](#)).

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.

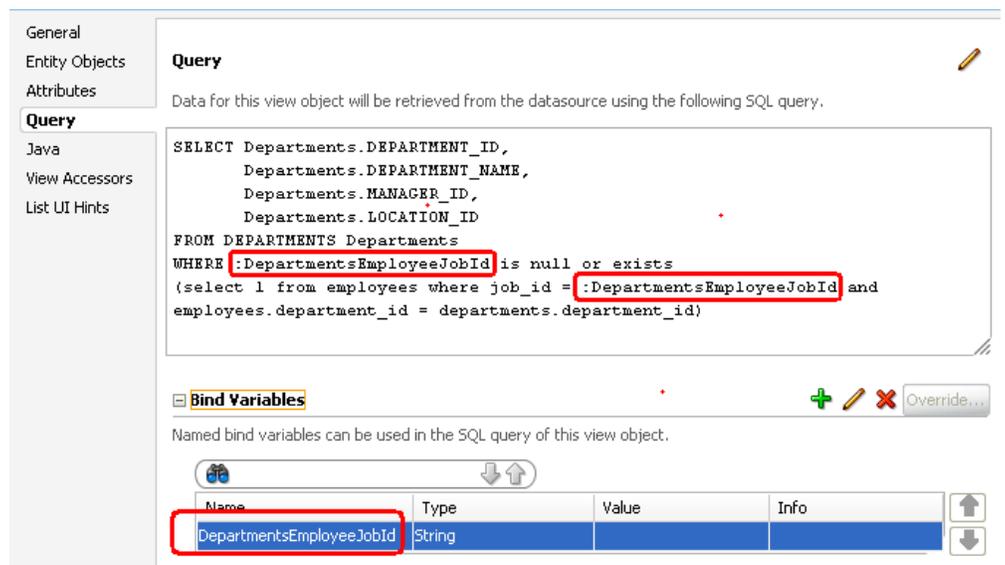
Let's use an example to illustrate this technique: we want to be able to search all departments that contain employees with a specific job. In other words, we want to add Employee JobId as a search item to the Departments group.

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!

7.1.8. Runtime Implementation of JHeadstart 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 adfc-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>searchDepartments</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.bean.JhsSearchBean</managed-bean-class>
  <managed-bean-scope>pageFlow</managed-bean-scope>
  <managed-property>
    <property-name>bindings</property-name>
    <value>#{'#{bindings}'}</value>
  </managed-property>
  <managed-property>
    <property-name>searchBinding</property-name>
    <value>#{'#{bindings.advancedSearchDepartments}'}</value>
  </managed-property>
  <managed-property>
    <property-name>searchItem</property-name>
    <value>DepartmentsEmployeeJobId</value>
  </managed-property>
  <managed-property>
    <property-name>dataCollection</property-name>
    <value>DepartmentsView1</value>
  </managed-property>
  <managed-property>
    <property-name>operators</property-name>
    <map-entries></map-entries>
  </managed-property>
</managed-bean>

```

You can see how the generic `JhsSearchBean` class is configured through the managed properties for usage in the Departments page. Related functionality like the maximum number of query hits allowed is 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()`.

7.1.8.2. 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`

7.1.8.3. 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.

Departments

Filter By

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'.

Departments |

Advanced Search

Result matches all conditions
 Result matches any condition
 Case Sensitive?

Department Id Location Id
 Department Name Special Ind
 Manager Id

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.

7.2. Configuring the Query

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

7.2.1. 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. Note that this property can only be used in combination with JHeadstart custom search.

7.2.2. Using Query Bind Parameters

Both groups and dynamic domains 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. You can specify a where clause with bind variables directly against the view object, or you can define bind parameters in a view criteria, and then apply this view criteria to the view object usage in the application module. The latter technique is usually faster because you can reuse an existing view object and associated view links.

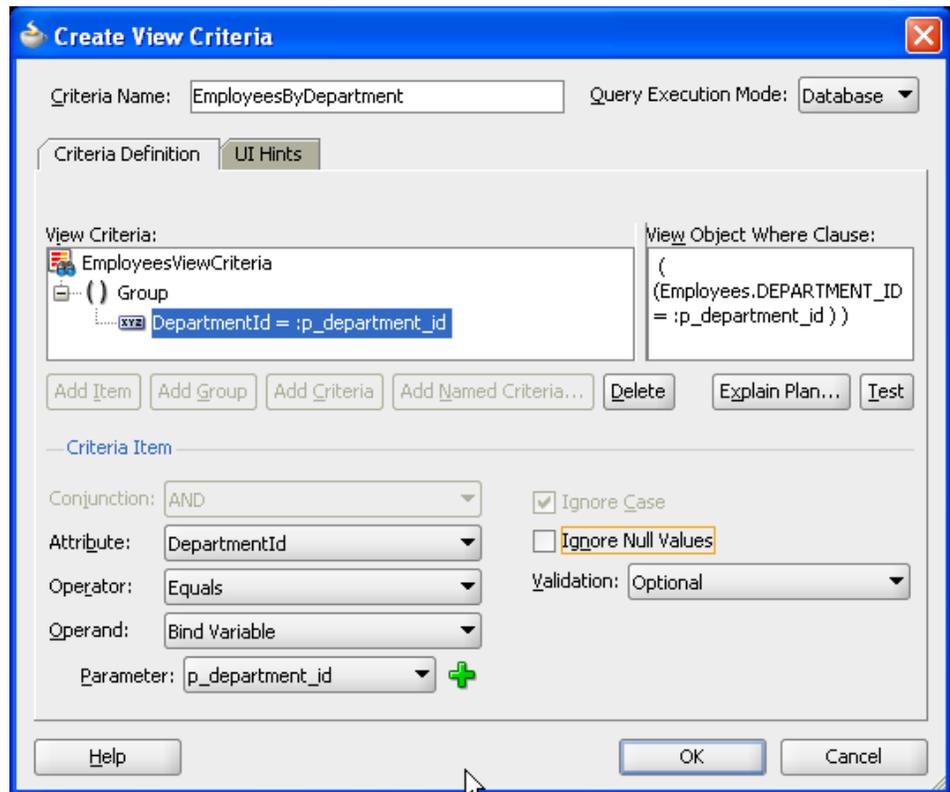
JHeadstart can at runtime pass values into these bind variables using the **Query Bind Parameters** property. For JHeadstart, it doesn't matter whether the bind variable is defined on the view object itself, or on a named view criteria applied on the view object usage.

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

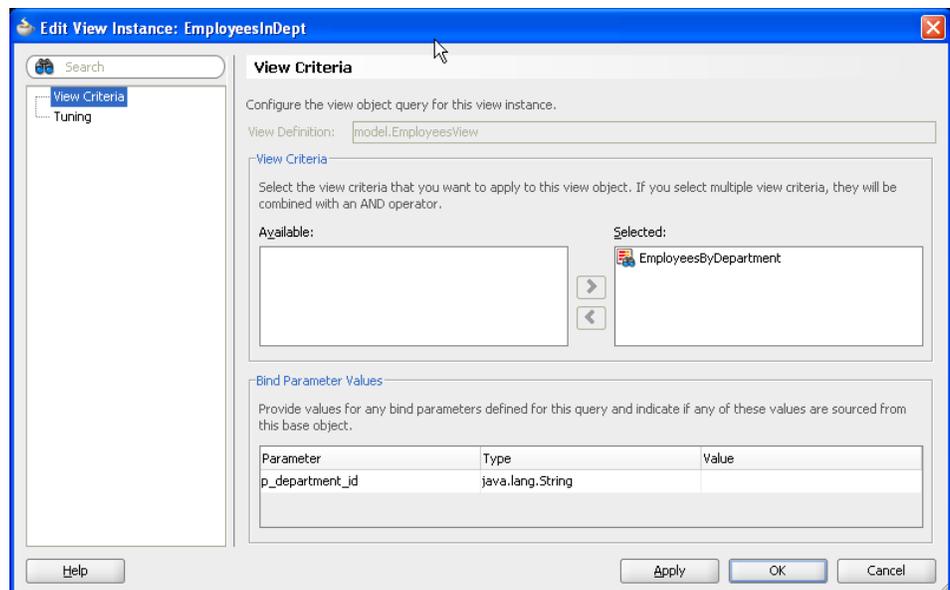
Base group is departments with **Layout Style**='form'. The **Display Type** of the item ManagerId is 'dropDownList'. We want the manager to be an employee of the department. The dropdown list should only contain employees that are in the department we are maintaining, in this case the department with DepartmentId=10. We use a dynamic domain to populate the ManagerId dropdown list

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

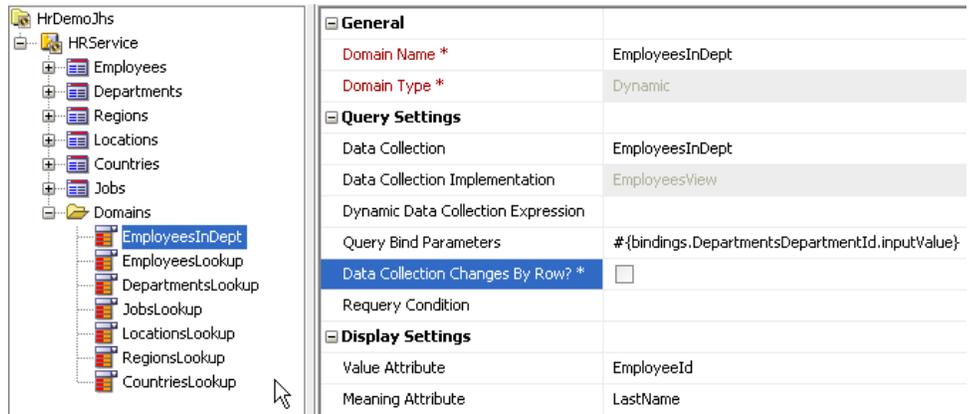
1. Go to your Model project, to the Employees view object and create a new named view criteria EmployeesByDepartment using a new bind variable p_department_id as shown below.



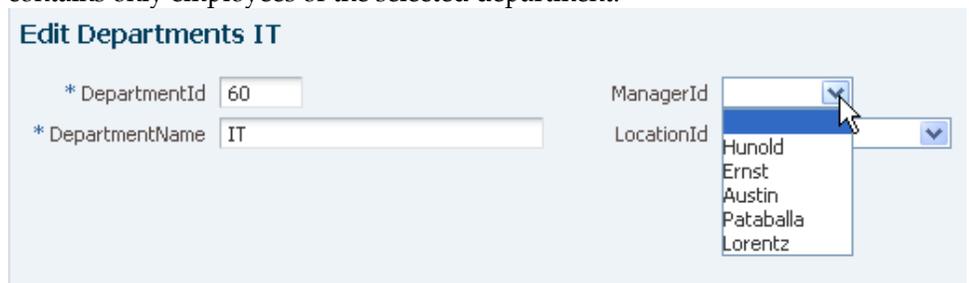
2. Add a new view object usage `EmployeesInDept` based on the `Employees` view object to the Data Model of the Application Module. Click the edit icon to apply the named view criteria `EmployeesByDepartment` that you just created.



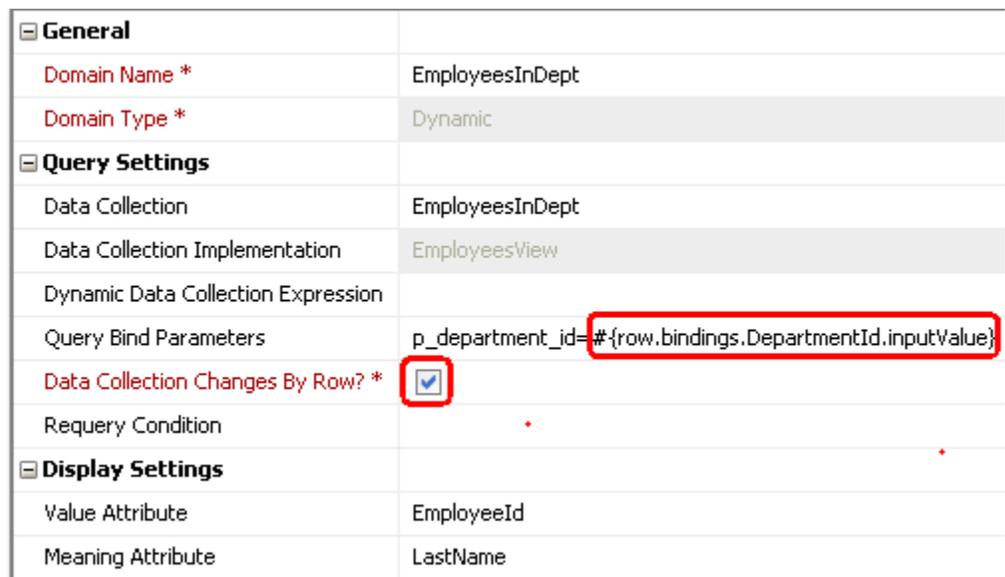
3. In JHeadstart, create a new **Dynamic Domain** based on the new view object usage and name it '`EmployeesInDept`'. In the **Query Bind Parameters** property enter this expression:
`p_department_id=#{bindings.DepartmentsDepartmentId.inputValue}`



4. Set the property **Domain** (on the `ManagerId` item) to `EmployeesInDept`.
5. Generate and run the application again. The dropdown list for `ManagerId` now contains only employees of the selected department.



Note that if you want to use this dropdown list in a table layout, you need to use a different value for **Query Bind Parameters** property, and you need to check the **Data Collection Changes By Row?** checkbox, as shown below.



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.

7.2.3. 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 adfc-config. If you specify query bind parameters for a dynamic domain in the Application Definition, JHeadstart generates such a managed bean in the domains adfc-config. Here is an example:

```
<managed-bean>
  <managed-bean-name>EmployeesInDeptQueryBindParams</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>namedParamDefs</property-name>
    <map-entries>
      <map-entry>
        <key>p_department_id</key>
        <value>#{'#{bindings.DepartmentsDepartmentId.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="invokeMethod"
  id="applyBindParamsEmployeesInDept"
  DataControl="HRServiceDataControl"
  InstanceName="HRServiceDataControl.dataProvider"
  MethodName="applyBindParams"
  ReturnName="HRServiceDataControl.methodResults.HRService
  IsViewObjectMethod="false">
  <NamedData MDName="voUsage" MDValue="EmployeesInDeptView"
  MDType="java.lang.String"/>
  <NamedData MDName="args"
  MDValue="#{EmployeesInDeptQueryBindParams.namedParams}"
  MDType="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. We use the

invoke action executable for this purpose. JHeadstart generates an invokeAction executable as follows:

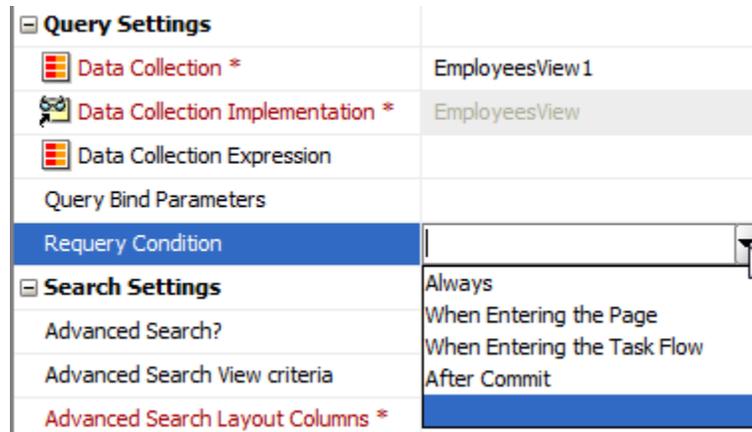
```
<invokeAction id="applyBindParamsEmployeesInDeptInvoke"  
             Binds="applyBindParamsEmployeesInDept" Refresh="always"/>
```



Reference: See the Javadoc or source of
`JhsApplicationModuleImpl.applyBindParams()`.

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 combo box property, the drop down list includes three common conditions that JHeadstart translates to a Boolean JSF EL expression, and 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 `#{jhsPageChanged}`. "jhsPageChanged" is a boolean request attribute that is set to true in `JhsNavigationHandlerImpl` class when the old JSF ViewRoot differs from the new ViewRoot.
- **When Entering the Task Flow:** the iterator binding is requeried when the user navigates to a new group task flow. This condition translates to the JSF expression `#{jhsTaskFlowChanged}`. "jhsTaskFlowChanged" is a boolean request attribute that is set to true in `JhsNavigationHandlerImpl` class when the old ADF task flow differs from the new task flow.
- **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}`. "jhsAfterCommit" is a boolean request attribute that is set to true in `CommitBean` class when the transaction is committed successfully.

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.

```
<action id="ExecuteQueryCountries" IterBinding="CountriesIterator"
        InstanceName="HRServiceDataControl.CountriesView1"
        DataControl="HRServiceDataControl" RequiresUpdateModel="true"
        Action="iteratorExecute"/>
```

- An invokeAction in the executables section. The refreshCondition of this invokeAction determines whether the executeQuery action binding is invoked or not.

```
<invokeAction id="ExecuteQueryCountriesInvoke" Binds="ExecuteQueryCountries"
              Refresh="renderModel" RefreshCondition="#{jhsPageChanged}"/>
```

This page is intentionally left blank.

Generating Transactional Behaviors

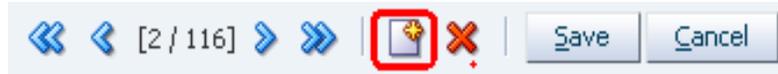
This chapter describes how you can influence the transactional behavior of generated pages. The properties in the **Operations** group in the Application Definition editor are used for this.

Operations	
Single-Row Insert allowed? *	<input checked="" type="checkbox"/>
Display New Row on Entry? *	
Single-Row Update allowed? *	<input checked="" type="checkbox"/>
Single-Row Delete allowed? *	<input checked="" type="checkbox"/>
Multi-Row Insert allowed? *	<input type="checkbox"/>
Multi-Row Update allowed? *	<input type="checkbox"/>
Multi-Row Delete allowed? *	<input type="checkbox"/>
Show Duplicate Row Button? *	<input checked="" type="checkbox"/>
Export Table Contents	
Table Editing Mode *	editAll

8.1. Enabling Insert Operation

8.1.1. Allowing Inserting Data in a Form Page

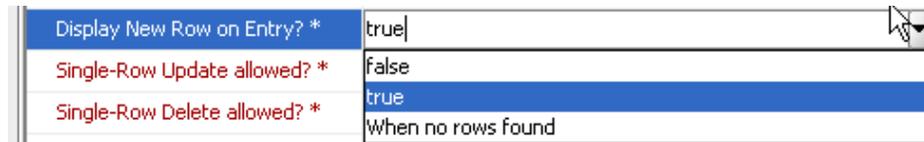
When checkbox **Single-Row Insert allowed?** is checked, JHeadstart will generate an iconic 'New' button on the page.



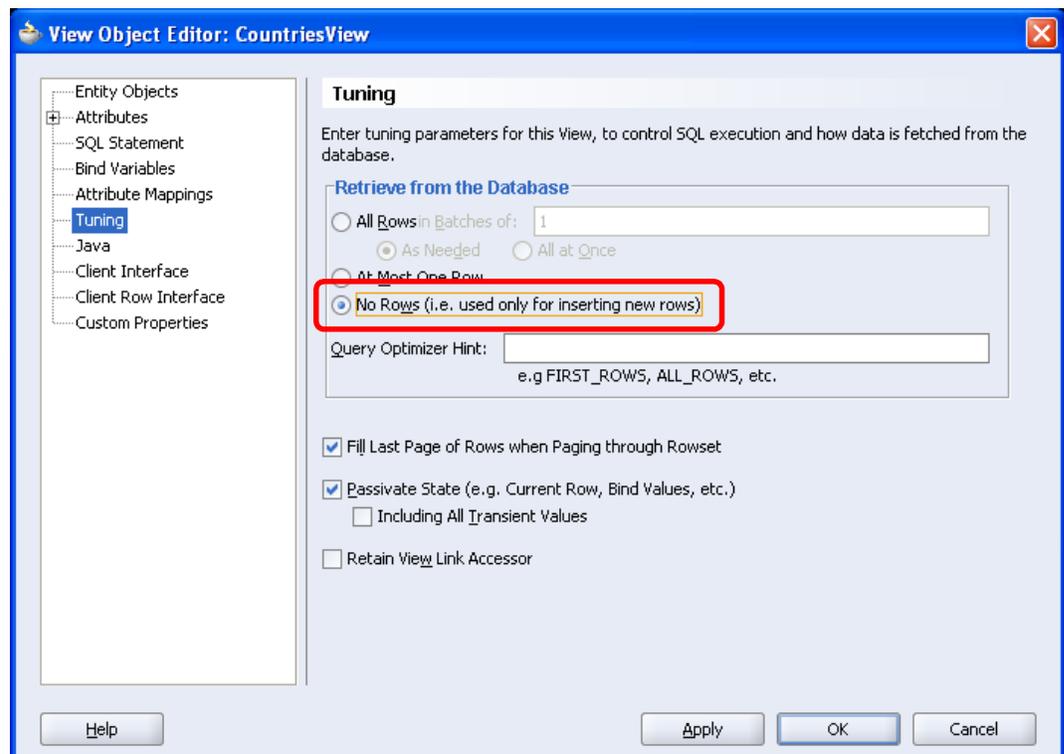
8.1.2. Building Insert Only Form Pages

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 property **Display New Row on Entry?** to "true".



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.



8.1.3. Entering in Insert Mode When No Rows Found

The easiest way to implement this behavior is by setting the property **Display New Row on Entry?** to “when no rows found”.

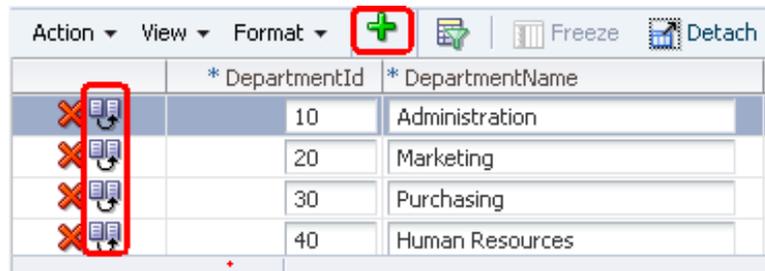
Another way to implement this behavior is to override the `executeQueryForCollection` method in your `ViewObjectImpl` class as follows:

```
protected void executeQueryForCollection(Object Object,
                                         Object[] bindParams, int i)
{
    super.executeQueryForCollection(Object, bindParams, i);
    if (getEstimatedRowCount()==0)
    {
        Row row = createRow();
        insertRow(row);
        row.setNewRowState(Row.STATUS_INITIALIZED);
    }
}
```

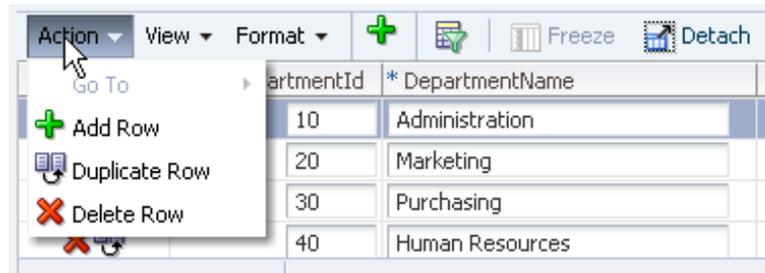
The difference is that with this approach, any default values you have set in the JHeadstart Application Definition Editor, are not applied. Note however that you can add custom Java code to the create method of an entity object to add (complex) defaulting logic.

8.1.4. Allowing the User to Insert Data in a Table Page

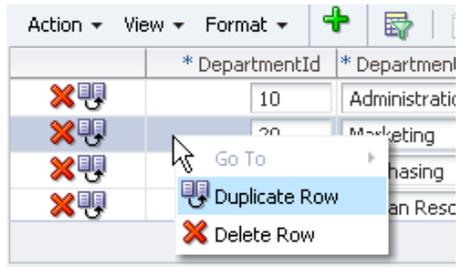
If the **Multi-Row Insert Allowed** checkbox is checked an iconic add Row button is generated in the header of the table. To quickly create new rows based on data of existing rows, you can check the **Show Duplicate Row Button** checkbox. An iconic button to duplicate a row is then generated in the first column of each row.



The same options are added to the Action drop down menu.



And the duplicate row function is also available on the popup menu accessible through a right-mouse-click on a row.



8.2. Enabling Update Operation

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.

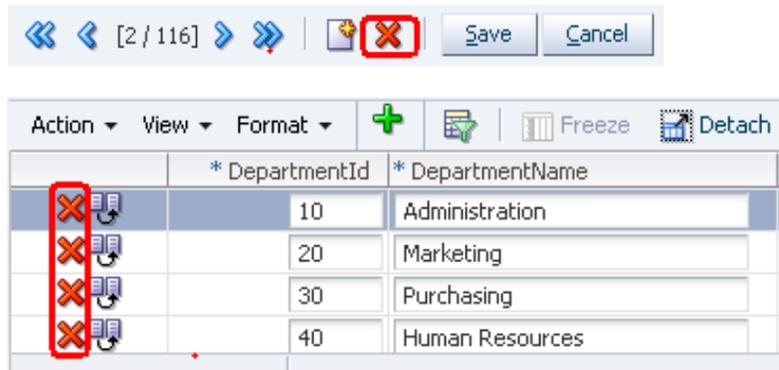
[-] Operations	
Update Allowed?	true
Disabled?	
[-] Validation	
Required?	true
Validator Binding	false
	while_new
	while_not_new

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 Operation

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 an iconic Delete button on a single row page, and the same iconic button in each row of a table where delete is allowed.



8.4. Conditionally Enabling Insert, Update and 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 “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_ASSISTANT 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')}
```

8.5. Runtime Implementation of Transactional Behaviors

8.5.1. Multi-Row Insert

To implement insert operations in a table, JHeadstart uses the class `oracle.jheadstart.controller.jsf.bean.TableBean`. The corresponding managed bean definition looks like this:

```
<managed-bean>
  <managed-bean-name>DepartmentsTable</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.bean.TableBean</managed-bean-class>
  <managed-bean-scope>pageFlow</managed-bean-scope>
  <managed-property>
    <property-name>name</property-name>
    <value>DepartmentsTable</value>
  </managed-property>
  <managed-property>
    <property-name>rangeBindingExpr</property-name>
    <value>#{ '#{bindings.DepartmentsTable}' }</value>
  </managed-property>
  <managed-property>
    <property-name>defaultValuesBean</property-name>
    <value>#{pageFlowScope.DepartmentsDefaultValues}</value>
  </managed-property>
  <managed-property>
    <property-name>focusItemId</property-name>
    <value>DepartmentsDepartmentId</value>
  </managed-property>
</managed-bean>
```

As you can see, the generic `TableBean` class is configured for specific usage in the `DepartmentsTable` using managed property settings.

When the user clicks the iconic Add Row button, the `addRow` method in this managed bean is called. This method performs the following functionality:

- creates the new row
- applies any default values if a default values bean is injected as managed property. This is the case when the **Default Display Value** property for one or more items in the group is set.
- discloses the row when the table is generated with property **Table Overflow Style** set to **Inline**. Note that this only works when the row has an auto-generated primary key attribute that is assigned in the `create()` method of the entity object. Otherwise, the key of the row is not yet known and the row cannot be disclosed.
- Sets the row as selected in the table.
- Puts the input focus on the first input item of the row.



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

8.5.2. Single-Row Insert

To implement insert operations in form layout, JHeadstart uses the class `oracle.jheadstart.controller.jsf.bean.CreateRowBean`. The corresponding managed bean definition looks like this:

```
<managed-bean>
  <managed-bean-name>CreateDepartments</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.bean.CreateRowBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>createBinding</property-name>
    <value>#{bindings.CreateDepartments}</value>
  </managed-property>
  <managed-property>
    <property-name>defaultValuesBean</property-name>
    <value>#{pageFlowScope.DepartmentsDefaultValues}</value>
  </managed-property>
</managed-bean>
```

The execute method in this bean is executed when the iconic ‘New ...’ button is pressed.

The execute method in the `CreateRowBean` first creates the new row using the Create binding in the page definition, then checks whether a default values managed bean is set as managed property. 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”.



Reference: See the Javadoc or source of `CreateRowBean`

8.5.3. Single-Row and Multi-Row Delete

To implement delete operations in form and table layout, JHeadstart uses the class `oracle.jheadstart.controller.jsf.bean.DeleteRowBean`. The corresponding managed bean definition looks like this:

```
<managed-bean>
  <managed-bean-name>DeleteDepartments</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.bean.DeleteRowBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>deleteBinding</property-name>
    <value>#{bindings.DeleteDepartments}</value>
  </managed-property>
  <managed-property>
    <property-name>commitBean</property-name>
    <value>#{jhsCommit}</value>
  </managed-property>
</managed-bean>
```

The execute method in this bean is executed when the iconic ‘Delete’ button is pressed. The execute method first passivates the state of the application module, then it deletes the row, and then it calls the `execute()` method on the injected `jhsCommit` bean (see section “Commit Handling”). 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 `DeleteRowBean`

8.5.4. Commit Handling

To implement commit operations, JHeadstart uses the class `oracle.jheadstart.controller.jsf.bean.CommitBean`. The corresponding managed bean definition looks like this:

```
<managed-bean id="_40">
  <managed-bean-name id="_41">SaveDepartments</managed-bean-name>
  <managed-bean-class id="_42">oracle.jheadstart.controller.jsf.bean.CommitBean</managed-bean-class>
  <managed-bean-scope id="_43">request</managed-bean-scope>
  <managed-property id="_44">
    <property-name id="_45">commitBinding</property-name>
    <value id="_46">#{bindings.Commit}</value>
  </managed-property>
  <managed-property id="_47">
    <property-name id="_48">successOutcome</property-name>
    <value id="_49">Commit</value>
  </managed-property>
</managed-bean>
```

The `execute()` method in this bean is executed when the 'Save' button is pressed, or when called from another bean like the `DeleteRowBean`. The `execute()` method performs 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.

8.5.4.1. Add Commit Behavior to a Custom Button

If you want to perform the same Commit handling on a custom button item that invokes a business method using the **Method Call** property, you can set the **Action** property to call the commit bean as follows:

```
#{SaveDepartments.execute}
```

If you want to have a customized transaction completed message, you can create a custom managed bean based on the `CommitBean` and set the `transactionCompletedMessageKey` property to the resource bundle key of your custom message. Then in the **Action** property, specify the execute method of this custom managed bean.

Alternatively, if you want to add commit behavior inside a custom managed bean method, you can add code like this (substitute `SaveDepartments` with the name of your generated task flow commit bean) :

```
CommitBean cb =
(CommitBean) JsfUtils.getExpressionValue("SaveDepartments");
cb.execute();
```

If you want to override the JHS-00100 and JHS-00101 message for this specific button, you can call methods to set another message key, or to set the message key to null to prevent a message being displayed altogether.

```
CommitBean cb =
(CommitBean) JsfUtils.getExpressionValue("SaveDepartments");
cb.setNoChangesMessageKey(null);
cb.setTransactionCompletedMessageKey("ACM-00004");
cb.execute();
```

See chapter 12, section 12.5.5 for more info on adding custom managed beans to JHeadstart-generated group task flows.



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

8.5.5. Rollback Handling

To implement rollback operations, JHeadstart uses the class `oracle.jheadstart.controller.jsf.bean.RollbackBean`. The corresponding managed bean definition looks like this:

```
<managed-bean>
  <managed-bean-name>jhsRollback</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.bean.RollbackBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>rollbackBinding</property-name>
    <value>#{bindings.Rollback}</value>
  </managed-property>
  <managed-property>
    <property-name>successOutcome</property-name>
    <value>Cancel</value>
  </managed-property>
</managed-bean>
```

The `execute()` method in this bean is executed when the 'Cancel' button is pressed, or when called from another bean like the `PendingChangesBean`. The `execute()` method simply executes the rollback binding.

Since the name of the Rollback binding is the same in all page definitions, the `jhsRollback` bean is only generated in `JhsCommon-beans.xml`. If you want to change the rollback behavior in your whole application, you can create a subclass of the `RollbackBean` and make a custom template for `JhsCommonBeans.vm` and change the managed bean class to the name of your subclass.

If you want to change rollback behavior in one specific task flow, you can add a custom managed bean by the same name ("jhsRollback") to the taskflow. The ADF Controller will first look for the request-scoped bean within the bounded taskflow, if not found, it will use the "default" `jhsRollback` bean defined in `JhsCommon-beans.xml`.

See chapter 12, section 12.5.5 for more info on adding custom managed beans to JHeadstart-generated group task flows.



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

Creating Menu Structures

JHeadstart supports two styles of creating menus:

- a static menu structure defined in XML format, generated based on the service and 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; the page template associated with a page will determine the actual *layout* of the menu. In this chapter, we will explain the two ways of creating a menu structure and how you can change the layout of the menu.

In the last paragraph, we will discuss the use of *dynamic* tabs that can be opened when clicking on a menu entry to create a multi-tasking user interface.

9.1. Static Menu Structure

By default, JHeadstart generates a static menu structure that reflects the structure of the services and groups as defined in the application definition. This static menu structure uses the Trinidad XML Menu Model facility. The XML Menu Model defines the menu hierarchy structure in one or more XML files. A managed bean that uses the class `org.apache.myfaces.trinidad.model.XmlMenuModel`, or a subclass, gets this XML Menu Model injected through a managed property named “source”.



Reference Web User Interface Developer's Guide for ADF. Section 20.7 Using a Menu Model to Create a Page Hierarchy.

http://docs.oracle.com/cd/E24382_01/web.1112/e16181/af_navigate.htm#CACIABAD

At application level, you can set the following properties to configure menu generation:

Menu Settings	
Allow Runtime Customization of Menu? *	<input type="checkbox"/>
Root Menu Model File *	/WEB-INF/menu_root.xml
Generate Root Menu Model File? *	<input checked="" type="checkbox"/>
Content First-Level Menu Tabs *	Base groups current service
Security	All services
Java	Base groups current service
	All base groups

Checking the **Allow Runtime Customization Menu?** Checkbox, will generate the dynamic, table-driven menu as discussed in the next section.

If you leave this checkbox unchecked, JHeadstart will generate a static menu structure. The **Content First-Level Menu Tabs** property determines which menu level is shown where:

- When set to its default of “Base groups current service”, then in the upper-right corner, also called the “menu global” area, menu items to switch services and to go to the home page will be generated, and the first level menu tabs will show the groups of the current service, as shown in the picture below:



- When set to “All Services”, the first level tabs will show the services and the home page, and second-level tabs will show the groups within the service, as shown in the picture below.



- When set to “All base groups”, the first level tabs will show a merged view of all groups across all services, as shown in the picture below.



The **Tabname** property of a group determines the label of the menu tab. When you check the application-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. 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**.

Customization Settings	
Add Menu Entry for this Group? *	<input checked="" type="checkbox"/>

Note that this property is only visible when you have switched the Application Definition Editor to expert mode

9.1.2. Customizing the Static Menu Structure

To understand how to customize the menu structure, it is useful to understand what files JHeadstart generates for you:

- a Root XML Menu as specified in the **Root Menu Model File** property. This file contains the home page item node, and references to all the service-level menu beans
- A managed bean “RootMenu” in JhsCommon-beans.xml which gets this Root XML Menu Model injected. This RootMenu bean is passed as an attribute to the page template of each page, causing the menu to render.
- For each service, an XML menu model file is generated as specified in the service-level property **Menu Model File**.

Menu Settings	
Menu Model File	/WEB-INF/menu-hrservice.xml
Generate Menu Model File? *	<input checked="" type="checkbox"/>
Menu Title	

This XML file defines the list of base groups as menu items.

- For each service, a '[ServiceName]Menu' managed bean is generated which gets the service XML Menu Model injected

If you want to change this generated menu structure, you can simply modify the generated XML files and uncheck the **Generate Menu Model File** checkbox. For example, we want to divide all base groups within the HR Service within two submenu's “People Management” and “Geographical Management” as shown below.

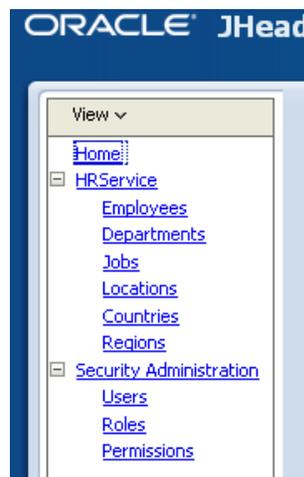


To do this, we can change the menu-hrservice.xml and group the items within two groupNode elements as shown below:

```
<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu">
  <groupNode id="HRService" idref="hr"
    rendered="{jhsUserRoles['HRService']}" label="HRService">
    <groupNode id="hr" idref="EmployeesMI" label="People Management">
      <itemNode id="EmployeesMI" label="Employees" action="uishell:Employees"
        focusViewId="Employees" rendered="{jhsUserRoles['Employees']}" />
      <itemNode id="DepartmentsMI" label="Departments"
        action="uishell:Departments" focusViewId="Departments"
        rendered="{jhsUserRoles['Departments']}" />
      <itemNode id="JobsMI" label="Jobs" action="uishell:Jobs" focusViewId="Jobs"
        rendered="{jhsUserRoles['Jobs']}" />
    </groupNode>
    <groupNode id="geo" idref="LocationsMI" label="Geographical Management">
      <itemNode id="LocationsMI" label="Locations" action="uishell:Locations"
        focusViewId="Locations" rendered="{jhsUserRoles['Locations']}" />
      <itemNode id="CountriesMI" label="Countries" action="uishell:Countries"
        focusViewId="Countries" rendered="{jhsUserRoles['Countries']}" />
      <itemNode id="RegionsMI" label="Regions" action="uishell:Regions"
        focusViewId="Regions" rendered="{jhsUserRoles['Regions']}" />
    </groupNode>
  </groupNode>
</menu>
```

9.1.3. Using a Static Menu Tree Layout

If you want to show the menu structure in a tree layout, you can simply switch the application-level Page Template to "JhsTreeMenuPageTemplate.jsf". The tree menu will look like this:



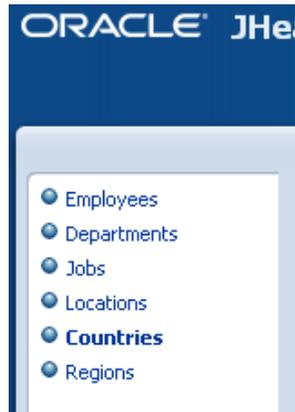
9.1.4. Customizing the Static Menu Layout

The XML Menu Model can be rendered using the `<af:navigationPane/>` element and the `<af:tree/>` element (as used in the previous section). The default JHeadstart Page template uses the `<af:navigationPane/>` element. The **value** property of this element can reference a managed bean that uses the XMLMenuModel class, as do the

RootMenu bean and service-level menu beans generated by JHeadstart. The **level** property of the navigationPane determines which level in the XML menu structure will be rendered. It will then stamp out the menu items for that level, with the layout as set in the **hint** property. The allowable values for the **hint** property are: **bar**, **buttons**, **choice**, **list** and **tabs**. The default JHeadstart page template uses **buttons** for the global menu, **tabs** for the first level menu, and **bar** for the second-level menu.

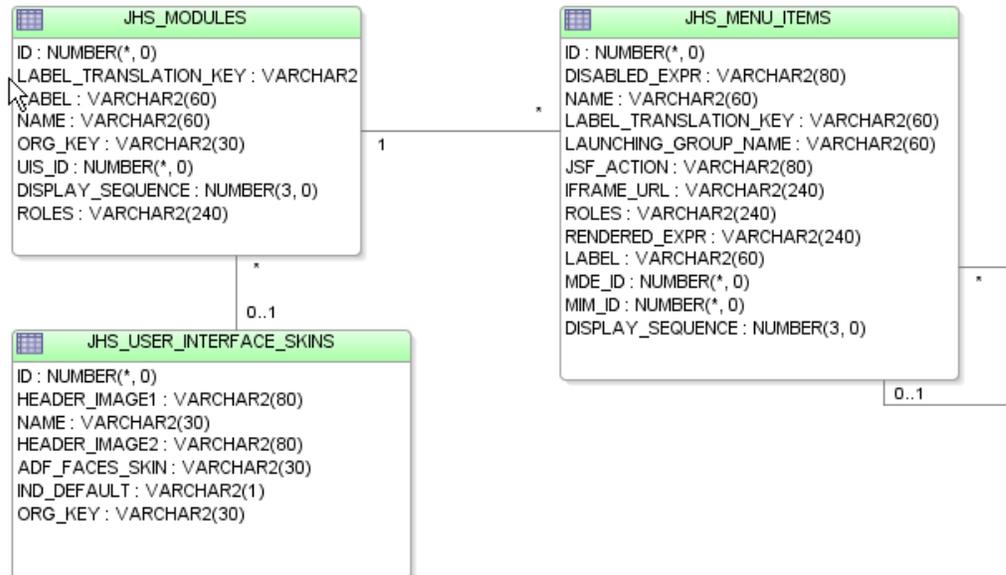
To change the layout hint used by the navigation panes, you can create your own custom page template, and set the application-level template to the name of your custom page template. If you use a layout hint that shows a specific level of menu items vertically, you probably want to display them at the left of the actual page content instead of at the top.

If you base your custom template on the the `JhsTreeMenuPageTemplate`, this can be achieved by replacing the `af:panelCollection` (which contains the `af:tree`) with the navigation pane. Setting the **hint** property to **list** will result in a menu level displayed like this:



9.2. Dynamic Menu Structure

Although the static menu structure can easily be changed, it requires application developers to change the menu XML files. If you do not want to depend on application developers to change the menu structure, then you should use the dynamic menu 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.

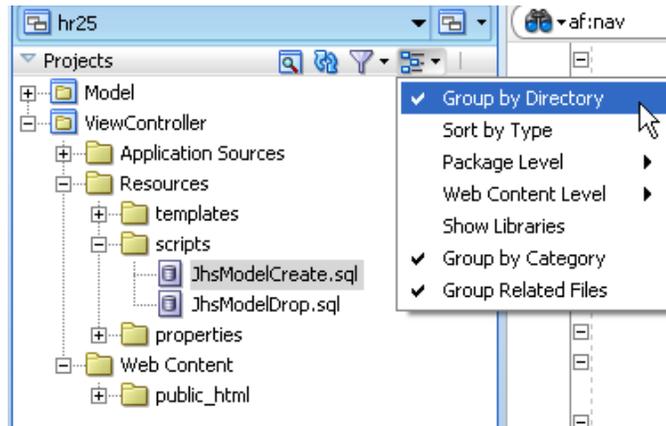


The top level of the menu structure is defined by so-called modules. A module can be seen as a logical subsystem of your application, and will often match with a service defined in one application definition, but this is not required. Each module has a nested structure of menu items. A menu item can 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. 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 check the "Group By Directory" option in the projects 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.

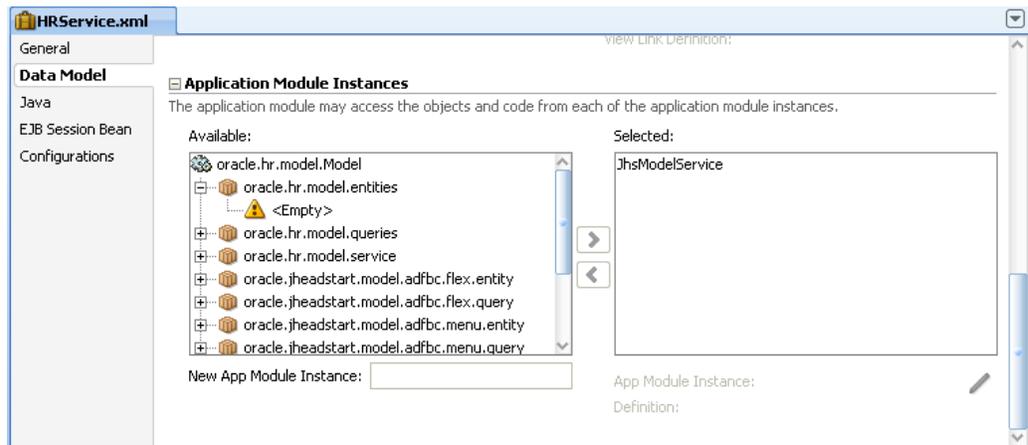
9.2.2. Enabling Dynamic Menus

To enable your application for use of dynamic menus, the first thing to do, is to check the application-level checkbox **Allow Runtime Customization of Menu?**.

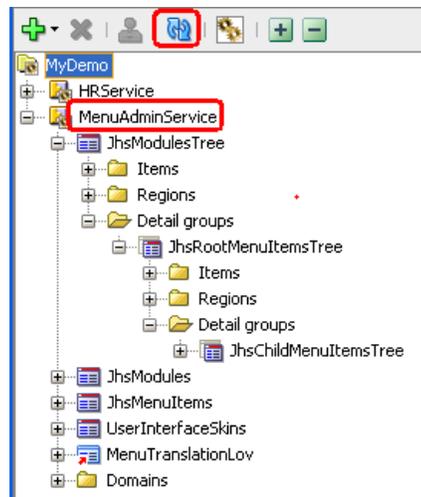
9.2.2.1. Running the JHeadstart Application Generator

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.

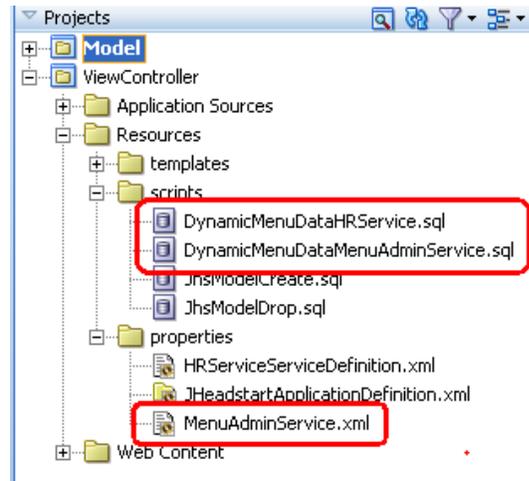


- An additional Service Definition file, named MenuAdminService.xml is generated. Click Save All after running the JAG and then in the JHeadstart Application Definition editor, select the application level node, and click the synchronize button in the toolbar to see the new MenuAdminService.



This service 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 losing these changes when you regenerate your “own” service definition. Note that like every other Service Definition, 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.

- A SQL Script named `DynamicMenuDataServiceName.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.

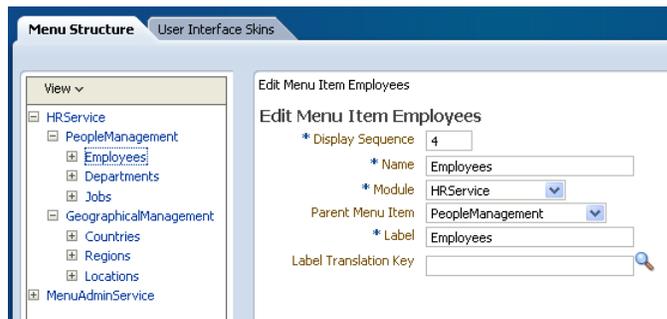
9.2.3. Defining the Menu Structure At Runtime

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 `HRService`, let's create only two top-level menu entries “`People Management`” with `Employees`, `Departments` and `Jobs` as level-two menu entries, and “`Geographical Management`”, with `Regions`, `Countries` and `Locations` underneath. We can do this by creating new menu items directly under the `HRService`, 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 HR Service using the module drop down list, the menu will look like this:

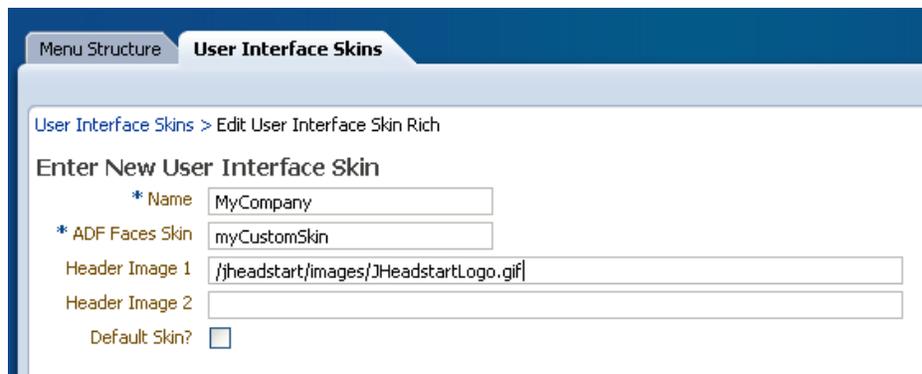


9.2.4. Using a Dynamic Menu Tree Layout

If you want to show the dynamic menu structure in a tree layout, you can simply switch the application-level **Page Template** to “JhsTreeMenuPageTemplate.jsf”.

9.2.5. 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.



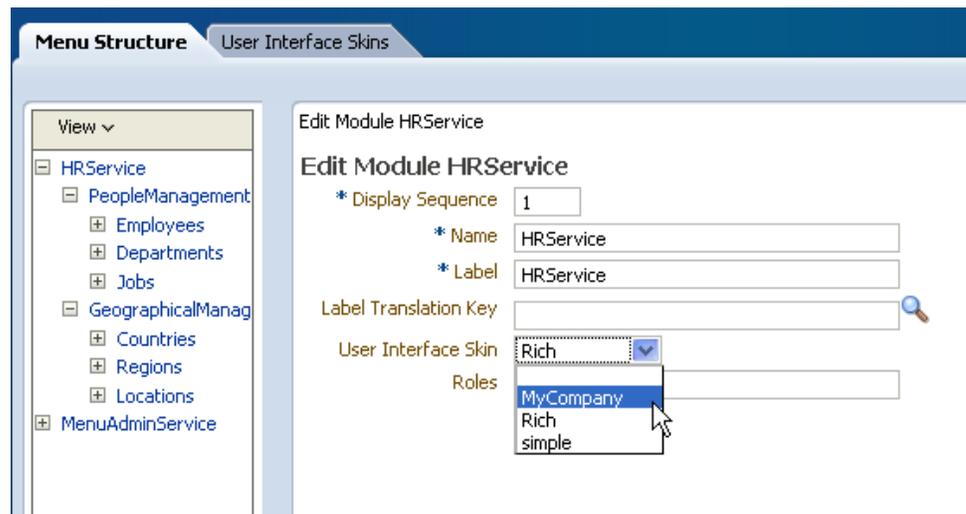
A JHeadstart User Interface Skin integrates with the ADF Faces Skinning feature as shown in the above screen shot. The value of the **ADF Faces Skin** property must be a value that exists in the `adf-faces-skins.xml` file, located in `web.xml`.



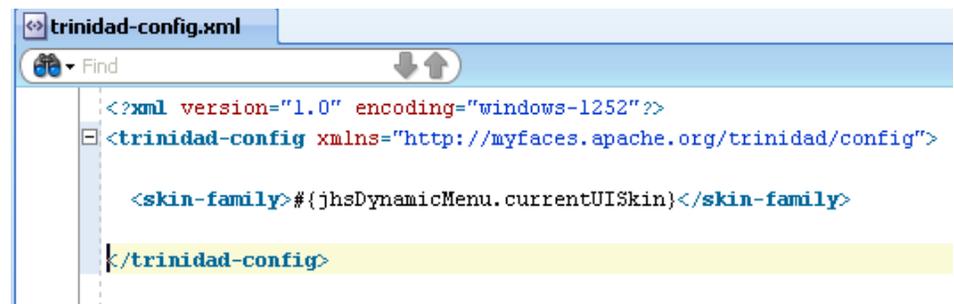
Web Link: More information about using ADF Faces Skinning and creating your own skin can be found in chapter 28 Customizing the Appearance Using Styles and Skins of the the Web User Interface Developers Guide for ADF:

http://docs.oracle.com/cd/E24382_01/web.1112/e16181/af_skin.htm#BAJFEFCJ

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 `trinidad-config.xml`, to configure ADF Faces to read the skin to use from the `jhsDynamicMenu.currentUISkin` managed bean property:



Modules that do not have a skin specified continue to be rendered with the default oracle skin.

9.3. Using Dynamic Tabs when Opening a Menu Item

Regardless of using a static or dynamic menu structure, you can choose to use dynamic tabs when clicking on a menu item. The menu itself is then displayed in tree layout, and clicking a menu item opens a dynamic tab. Clicking another menu item will open another tab, leaving the first tab open. By using dynamic tabs, you effectively create a multi-tasking user interface within your ADF application. Each tab has its own transaction, so a user can temporarily switch to another tab, and then return to his original task by going back to the first tab.



JHeadstart provides full support for this dynamic tabs user interface pattern, including:

- A dynamic tabs page template that works with both an XMLMenuModel based tree menu, as well as the dynamic table-driven tree menu.
- A JSF phase listener that will automatically mark the current tab dirty, showing the tab label in italics, if there are pending changes in the tab task flow
- The option to initially display one or more dynamic tabs at application start-up
- The option to launch an additional dynamic tab within a group task flow displayed under the current dynamic tab.
- A tab unique identifier that will be used to re-select an already open tab if the user clicks a menu item or other button that launches the tab.
- The option to close a tab through the close icon on the tab, or through some command action in the task flow that closes the current tab. You can also configure a tab to not being closeable.

The next sections will describe these features in more detail.

9.3.1. Enabling Dynamic Tabs

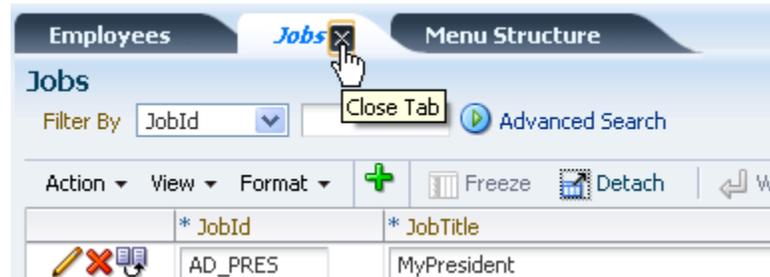
You enable dynamic tabs by setting the application-level **Page Template** property to `/common/pageTemplates/JhsDynamicTabsPageTemplate.jsf`.

Since each task flow displayed under a tab must have its own transaction scope, you also need to set the application-level property **Data Control Scope** to `isolated`. Furthermore, all tab task flows are displayed as a region, so you cannot use dynamic tabs in combination with stand-alone pages. So make sure you leave the application-level property **Default group Usage** to its recommended default of `Region with Page Fragments`. Since the dynamic tabs are managed at the UI Shell page level, you also should leave the property **Default group Region Access** to its recommended default of `Common UI Shell Page`.

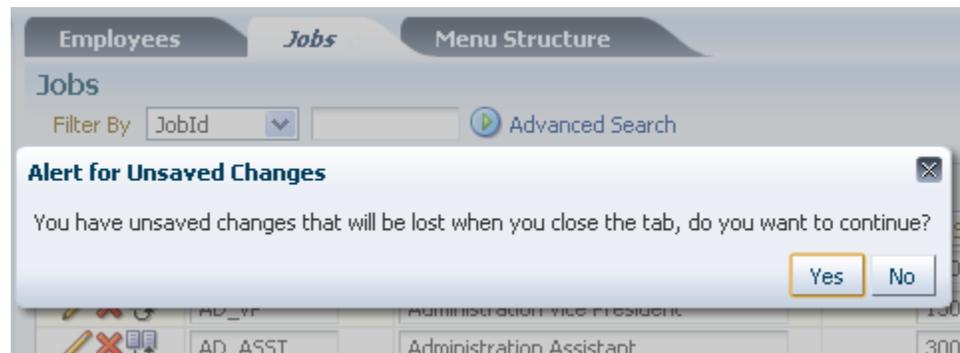
If you have created your own customized dynamic tabs page template, and have set the Page Template property to your custom page template, you can still use the dynamic tabs functionality. In this case you need to check the application property **Use Dynamic Tabs?**.

9.3.2. Marking the Current Tab Dirty

If you made changes to the data displayed within a dynamic tab, then the moment a request is sent to the server, this tab will be marked dirty which results in the tab label being displayed in italics.



In addition, if you close a tab that is marked dirty, you will get a dialog that warns you about pending changes that might be lost.



The functionality to mark the current tab dirty is implemented in `faces-config.xml`, where the Jheadstart Phase Listener is configured that performs this functionality.

```
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  <lifecycle>
    <phase-listener>oracle.jheadstart.controller.jsf.listener.JhsPhaseListener</phase-listener>
  </lifecycle>
</faces-config>
```

So, if you want to switch off this functionality for whatever reason, you create a custom template for `facesConfig.vm` and comment out the lines that generated the phase listener. See chapter 12 Customizing Generator Output for more information on using custom templates.

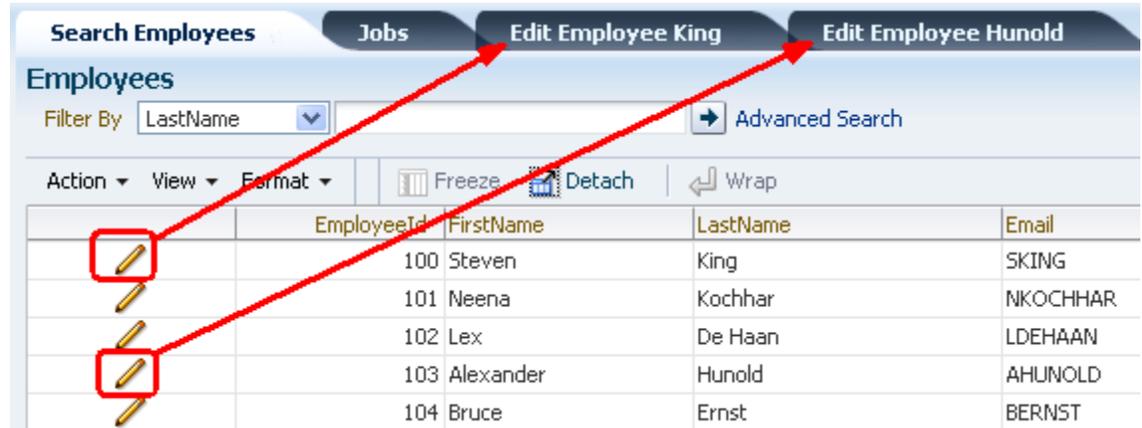
9.3.3. Displaying Initial Tabs at Startup

You can also choose to display one or more dynamic tabs at application start up, for example tabs that allow you to work with the key domain objects of your application. To initially display a dynamic tab with a group task flow, you check the group-level property **Show as Initial Dynamic Tab?**. If you do this, you still have the option to show a menu entry for this tab. If you don't want the menu option to appear, you can uncheck

the group property **Add Menu Entry for this Group?**. If you do this, the initial tab will not be closeable anymore, as there is no menu item anymore to re-open it.

9.3.4. Launching a New Tab from the Current Tab Task Flow

A common user interface pattern in combination with dynamic tabs is a search task flow that displays search results of some key domain object in a table, and then opening a new tab to work with one of the rows from the search result, as shown below.



JHeadstart fully supports this pattern. You can create a link, or iconic button that launches another group in a new dynamic tab. For detailed steps on how to implement this pattern see chapter 6 section 6.14 "Navigating Context-Sensitive to a group task Flow (Deep Linking) "

9.3.5. Closing a Dynamic Tab

You can close a tab by clicking on the close icon of the tab. A tab of a group that has property **Show as Initial Dynamic Tab?** checked and property **Add Menu Entry for this Group?** unchecked is not closeable, as there is no menu item to re-open it.

You can also close the current tab from within the group task flow displayed in the tab. This is useful if there is a clear last step that when performed by the end user should close the tab. To generate a button that closes the tab, you can define an unbound item in the group with the **Display Type** property set to a value that renders as an (iconic) button or hyperlink. Then, set the **Action** property of the item to `CloseCurrentTab`. You can set the **Action Listener** property to a custom managed bean method that should be executed prior to closing the tab. If you conditionally want to close the current tab based on the success of a last action performed, then set the **Action** property to a custom managed bean method that conditionally returns the string `CloseCurrentTab` or `null` when the tab should not be closed.

If you want to close the current tab when clicking one of the standard-generated buttons like Save or Cancel, you can create a custom template to make sure the `CloseCurrentTab` action outcome is used.

See chapter 12 "Customizing Generator Output" for more information on creating custom templates and custom managed beans.



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, page fragments, and bounded taskflows, 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, taskflows 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 explains the security features JHeadstart adds to standard ADF security.

10.1. Understanding and Choosing Security Options with ADF and JHeadstart

The security features in ADF 11 have strongly improved over previous versions. Oracle ADF Security is built on top of the Oracle Platform Security Services (OPSS) architecture, which itself is well-integrated with Oracle WebLogic Server. While other security-aware models exist that can handle user login and resource protection, Oracle ADF Security is ideally suited to provide declarative, permission-based protection for ADF bounded task flows, top-level web pages that use ADF bindings (pages that are not contained in a bounded task flow), and at the lowest level of granularity, rows of data defined by ADF entity objects and their attributes.



The remainder of this chapter assumes you are familiar with Fusion Developer's Guide for ADF, **chapter 35 Enabling Security in a Fusion Web Application**:

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/adding_security.htm#BGBGJEAH

In this paragraph we briefly discuss the various security settings you can make in the JHeadstart Application Definition editor (at Application level). Subsequent paragraphs explain in more detail what happen with each setting.

Security	
Authentication Type	ADF/JAAS
Authorization Type	ADF/JAAS
Secure All Pages? *	<input checked="" type="checkbox"/>
Authorize Using Group Permissions? *	<input checked="" type="checkbox"/>
Role/Permission Prefix	
Administrator Role	ADMIN
User Role	USER
Insert Allowed EL Expression	<code>#{jhsUserRoles['\$GROUP_NAME\$.create']}</code>
Update Allowed EL Expression	<code>#{jhsUserRoles['\$GROUP_NAME\$.update']}</code>
Delete Allowed EL Expression	<code>#{jhsUserRoles['\$GROUP_NAME\$.delete']}</code>

10.1.1. Authentication and Authorization Type

We recommend to set the **Authentication Type** to ADF/JAAS. By using ADF/JAAS you can leverage all of the standard ADF security features as explained in chapter 29 of the Fusion Developer's Guide, and it allows you to use the runtime customizations supported by ADF Faces, like reordering table columns and adding saved searches.

If you need to secure application resources based on user roles, then also set the add **Authorization Type** property to 'ADF/JAAS'. You can then secure the ADF Security-aware resources: bounded taskflows, page definitions of pages in the unbounded taskflow, entity objects and entity object attributes.

If you want to generate your application with ADF/JAAS security settings, you first need to run the ADF Security Wizard. If you forget this, you will get an error message upon generation.

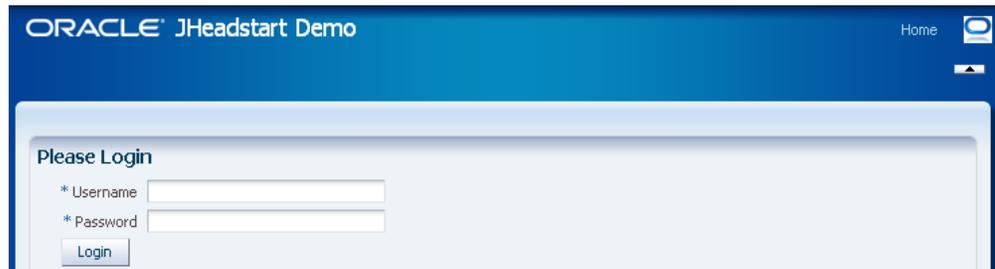
The “custom” authentication option is intended for backwards compatibility. The “custom” authentication option will use the JHeadstart security tables (or your own security tables) to authenticate against.

Note that if you do want to authenticate against database tables, you can use a JAAS Custom Login Module in combination with ADF/JAAS authentication. See the documentation of your application server for more information on writing JAAS custom login modules or using predefined login modules provided by the application server.

If you do not need to secure the ADF security-aware resources, but only want to show/hide application functions based on user roles and/or group permissions (see below), then set the **Authorization Type** to “custom”.

10.1.2. Secure All Pages

If you check this option, you will get a login page when trying to access the application.

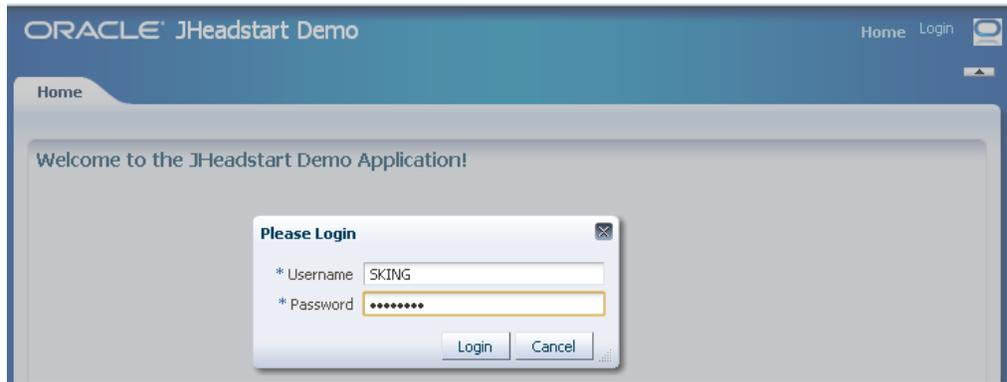


The application redirects to the login page using the “allPages” web resource collection in the web.xml:

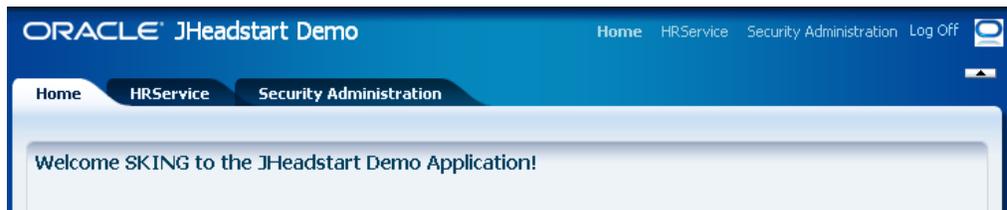
```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>adfAuthentication</web-resource-name>
    <url-pattern>/adfAuthentication</url-pattern>
  </web-resource-collection>
  <web-resource-collection>
    <web-resource-name>allPages</web-resource-name>
    <url-pattern>/faces/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>valid-users</role-name>
  </auth-constraint>
</security-constraint>
```

JHeadstart will add this allPages web resource collection if it is not already present. **If you uncheck this checkbox, you need to manually remove the “allPages” web resource collection in the web.xml if present.** This can happen when you ran the ADF security wizard in “ADF Authentication” mode, or you generated the application before with this checkbox checked.

If you generate your application with this checkbox unchecked, then JHeadstart will generate the page template with an additional “login” link and login popup that allows the user to access the secure parts of the application.



After the user logged in, all secured application functions the user has access to will be displayed, as well as a log out link.



10.1.3. Authorize Using Group Permissions

When you add security to your application, you need to check whether a user is authorized to perform some application function. 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 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.

See section [Restricting Access to Groups based on Authorization Information](#) for more information.

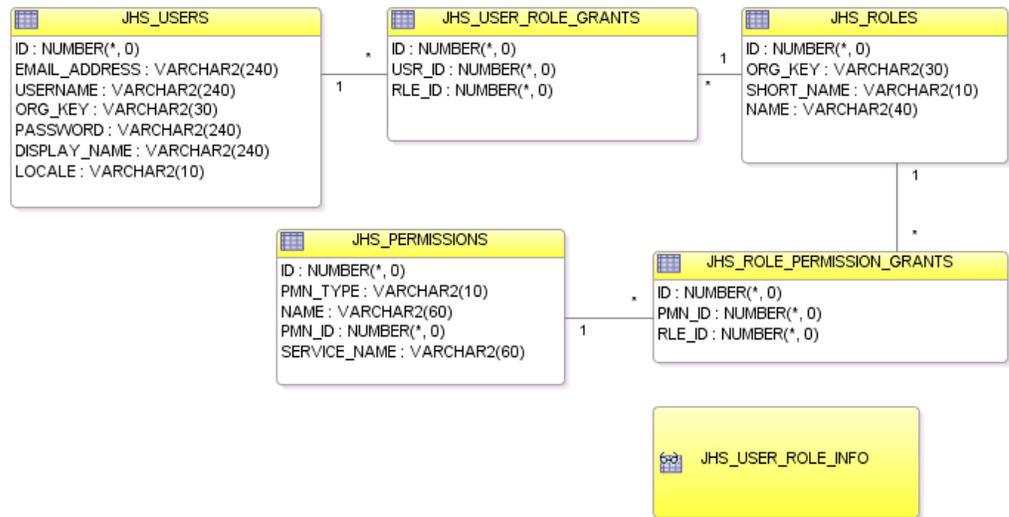
10.1.3.1. JHeadstart Group Permissions and ADF Resource Permissions

If you enabled ADF Authorization in the ADF Security wizard, you are already familiar with the concept of permissions. For example, you can grant the view permission on a bounded taskflow to a specific role. ADF resource permissions and JHeadstart group permissions partly serve the same purposes, although there are some differences as well:

- Both ADF and JHeadstart permissions can be used to prevent access to a bounded taskflow or page definition in an unbounded taskflow. When trying to access an unauthorized resource, an error message is thrown.
- Both ADF and JHeadstart permissions can be used to hide application functions (menu tabs, buttons, links, etc.) in the user interface that provide access to unauthorized ADF resources.
- JHeadstart permissions can be used to switch off insert and/or update and/or delete capabilities in a page. See section [Restricting Group And Item Operations based on Authorization Information](#) for more info.
- JHeadstart uses database tables to store the permissions and the roles that have grants to these permissions. Maintenance screens can easily be generated to maintain the security information, as explained in the next section.
- ADF resource permissions and associated roles can be maintained through the jazn-data.xml file at design time in JDeveloper. Once deployed, you can use tools like Oracle Enterprise Manager to maintain security information. See chapter 29 of the Fusion Developers Guide for more information.

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.

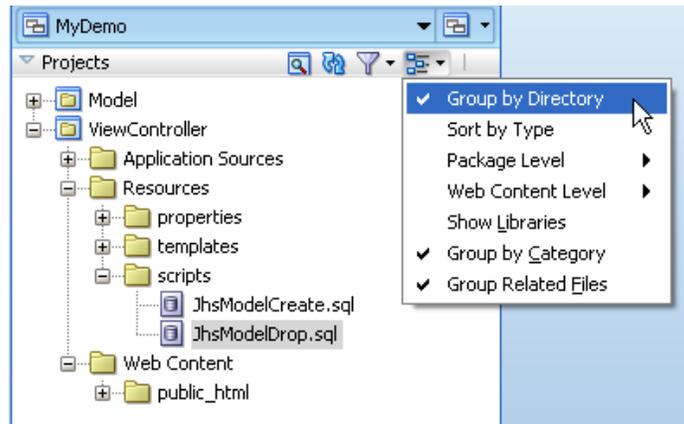


Depending on the security settings you have made, you will use all, some or none of the above database objects:

- When **Authentication Type** is set to “ADF/JAAS”, and **Authorization Type** is set to “ADF/JAAS” and checkbox **Authorize Using Group 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 **Authorization Type** is set to “Custom” or “ADF/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 Group by Directories option 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.

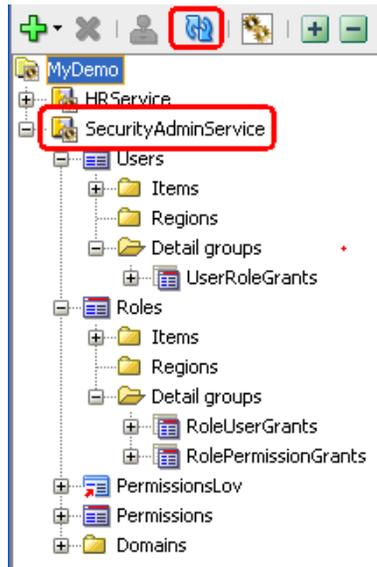
 **Attention:** The JhsModelCreate.sql script creates database tables for all table-driven JHeadstart runtime features. Additional tables for dynamic menus, translations and flex items are also created. If you do not plan to use these other features you can create your own script that only creates the 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 service definition SecurityAdminService.xml is generated that can be used to generate the security administration screens. Click Save All after running the JAG and then in the JHeadstart Application Definition editor, select the application level node, and click the synchronize button in the toolbar to see the new SecurityAdminService.



This service 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 losing these changes when you regenerate your “own” service definition.

As you can see in the above screen shot, the generated SecurityAdminService 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** “ADF/JAAS” and **Authorization Type** “ADF/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 SecurityAdminService.

10.3. Using ADF/JAAS for Authentication

When you run the JHeadstart Application Generator with service-level property **Authentication Type** set to “ADF/JAAS”, the following happens:

- ADF security settings in the web.xml might be modified if needed.
- A login page and associated login bean is generated. The login page can be a full page, or a popup accessible from a login link. This is determined by property **Secure All Pages** as explained before.
- A logout button is generated.
- Default users and roles are defined in jazn-data.xml.

These actions are discussed below in more detail.

10.3.1. Changes in web.xml

To ensure the JHeadstart-generated login page is used, the authentication method is set to “FORM” and the form-login-page is set to the generated JHeadstart login page.

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>jazn.com</realm-name>
  <form-login-config>
    <form-login-page>/faces/security/pages/Login.jspx</form-login-page>
    <form-error-page>/loginErrorServlet</form-error-page>
  </form-login-config>
</login-config>
```

The following init parameters are set in the adfAuthentication servlet:

```
<servlet>
  <servlet-name>adfAuthentication</servlet-name>
  <servlet-class>oracle.adf.share.security.authentication.AuthenticationServlet</servlet-class>
  <init-param>
    <param-name>success_url</param-name>
    <param-value>/faces/pages/Home.jspx</param-value>
  </init-param>
  <init-param>
    <param-name>end_url</param-name>
    <param-value>/faces/pages/Home.jspx</param-value>
  </init-param>
  <init-param>
    <param-name>allow_success_url_param_overwrite</param-name>
    <param-value>>true</param-value>
  </init-param>
  <init-param>
    <param-name>allow_logout_url_param_overwrite</param-name>
    <param-value>>true</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

The parameters ensure that by default the user is redirected to the home page after login and logout, however, this destination page can be overridden by including success_url or end_url as request parameter in the login or logout URL.

10.3.2. 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 losing 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 ADF/JAAS authentication, this method uses a Weblogic specific JAAS implementation to authenticate the user, similar to what is documented in the Fusion Developer's Guide.

By using this JAAS API, 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.

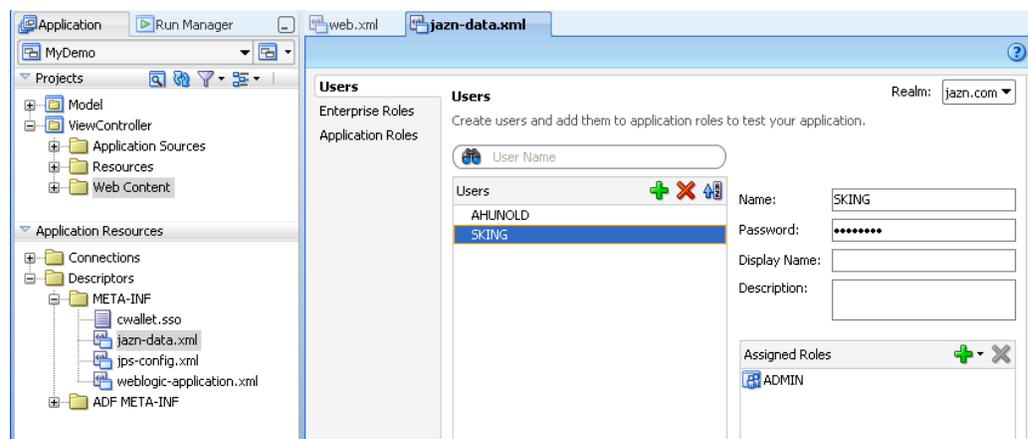
10.3.3. Logout Button

Using the `/default/misc/file/menuGlobal.vm` template, called from the `default/misc/file/fileGenerator.vm` template, a logout link is generated in the global menu area. The logout button redirects to the `adfAuthentication` filter which takes care of the log out process.

10.3.4. Default Users and Roles in jazn-data.xml

JHeadstart adds default users and roles in the `jazn-data.xml` file in the `META-INF` directory under the Descriptors folder in the Application Resources pane.

The `jazn-data.xml` specifies two default users, SKING and AHUNOLD with password "welcome1", the Administrator role assigned to SKING and the user role assigned to AHUNOLD.



10.4. Using Custom Authentication

When you run the JHeadstart Application Generator with service-level property **Authentication Type** set to “Custom”, the following happens:

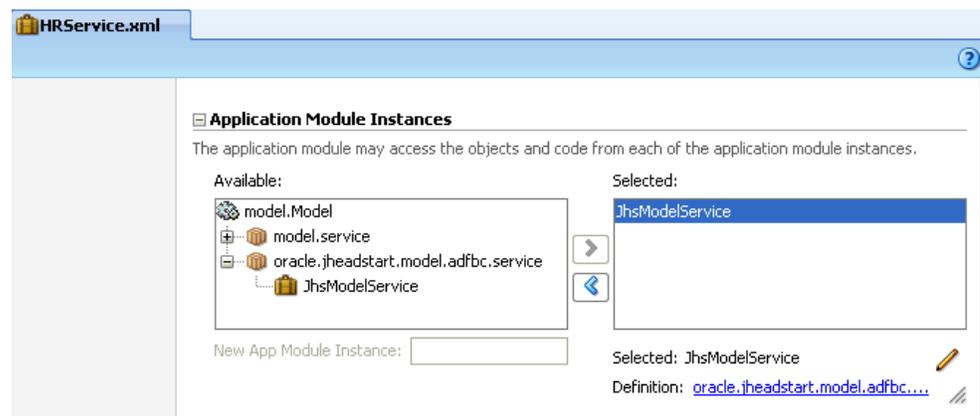
- The JHeadstart authentication servlet filter is configured in the web.xml
- JhsModelService application module is added as a nested application module to your application module.
- A login page and associated login bean is generated.
- A logout button is generated
- SQL script createSampleUsersAndRoles.sql is generated. See section [Sample Users And Roles](#) for more information.
- The SecurityAdminAppDef application definition file is generated. See section [Generating Security Administration Pages](#) for more information.

10.4.1. 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 is specified as filter parameter.

10.4.2. 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.



10.4.3. 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 losing 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`.

10.4.4. 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`. However, this URI is configured in the JHeadstart Authentication filter as the logout URL, see section [JHeadstart Authentication Filter](#) for more information.

10.5. Restricting Access to Groups based on Authorization Information

When you have 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.

[-] Authorization	
Authorized Roles/Permissions	ADMIN, HR_MANAGER

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.

10.5.1. Restricting Group Access using Permissions

The above example used role names to restrict access to a particular group. As explained in section [Authorize Using Group Permissions](#) 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**.

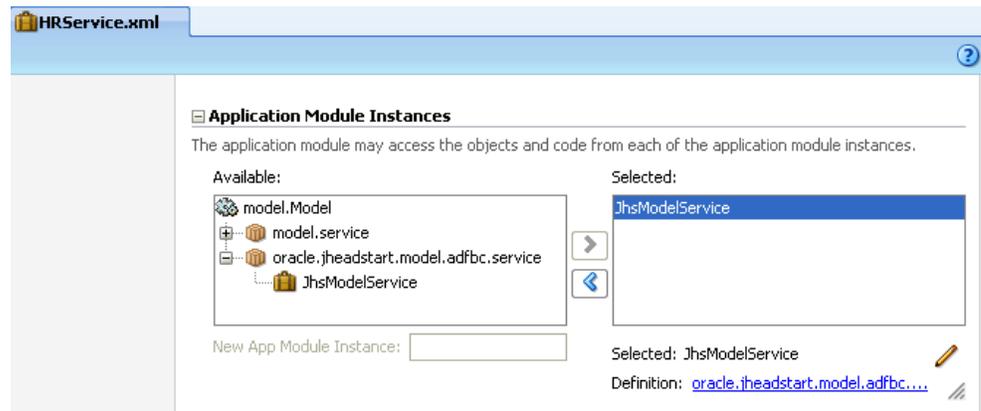
When you generate your application with this setting, the following happens

- 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.

10.5.2. 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 group permissions. Note that by creating `JhsModelService` as a nested application module, it will inherit the database connection of the parent application module.



10.5.3. 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 (ADF/JAAS or custom security).

To accomplish this, the JHeadstart runtime includes a class called `JhsAuthorizationProxy`. If you have set the Application-level property **Authorization Type** 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 whenever the application needs authorization information. So whether ADF/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.

When group permissions are used in combination with ADF/JAAS-based authorization, this class 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. Note that all authorizations are cached for the duration of the session.



Reference: See the Javadoc of `JhsAuthorizationProxy`.

10.5.3.1. 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.

10.5.3.2. Accessing the Authentication Proxy in the Controller layer

As mentioned before, JHeadstart also performs a roles check in the JSF PageLifecycle. This is 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/uiomodel"
  <parameters>
    <parameter id="roles" value="admin,manager"/>
  </parameters>

```



Reference: See the Javadoc of the methods `prepareModel ()` and `checkRoles ()` in the `JhsPageLifecycle` class.

10.6. 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	<code>#{jhsUserRoles['HR_MANAGER']}</code>
Update Allowed EL Expression	<code>#{jhsUserRoles['HR_MANAGER,HR_ASSISTANT']}</code>
Delete Allowed EL Expression	<code>#{jhsUserRoles['HR_MANAGER']}</code>

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.

10.6.1. 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? *	<input checked="" type="checkbox"/>
Authorization Type	JAAS
Authorize Using Group Permissions? *	<input checked="" type="checkbox"/>
Role/Permission Prefix	
Administrator Role	ADMIN
User Role	USER
Insert Allowed EL Expression	<code># {jhsUserRoles['\$GROUP_NAME\$.create']}</code>
Update Allowed EL Expression	<code># {jhsUserRoles['\$GROUP_NAME\$.update']}</code>
Delete Allowed EL Expression	<code># {jhsUserRoles['\$GROUP_NAME\$.delete']}</code>
When Access Denied go to Next Group? *	<input checked="" type="checkbox"/>

10.6.2. 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? *	<code># {jhsUserRoles['admin, hrmanager']}</code>
Display in Table Layout? *	false
Display in Table Overflow Area? *	<code># {jhsUserRoles['admin, hrmanager']}</code>

- Enabledness is determined using the **Disabled** property
- Updateability is determined using the **Update Allowed?** property

Operations	
Update Allowed?	<code># {jhsUserRoles['hrmanager']}</code>
Disabled	<code># {jhsUserRoles['admin']}</code>

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.

10.7. 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

10.7.1. 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`**.

10.7.2. 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.

10.7.3. Changes to SQL Script Templates

JHeadstart uses the following templates to generate entries into the various security tables:

- `CREATE_SAMPLE_ROLES_AND_USERS`:
default/misc/file/createSampleUsersAndRoles.vm
- `JHS_PERMISSIONS_DATA`:
default/misc/file/jhsPermissionsdata.vm

You can make custom templates to generate and execute your custom SQL script. See chapter 12 "Customizing Generator Output" for more information on creating custom templates.

This page is intentionally left blank.

Internationalization and User Assistance

This chapter discusses the JHeadstart support for multiple languages, as well as various options to provide user assistance.

The following topics are discussed:

- National Language Support (NLS) in JHeadstart
- Using a Database Table to Store Translatable Strings
- Runtime Implementation of National Language Support
- Error Reporting
- Outstanding Changes Warning
- Using Online Help
- Using Function Keys

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.

☐ Internationalization	
NLS Resource Bundle *	mydemo.view.ApplicationResources
Resource Bundle Type *	databaseTable
Override NLS Resource Bundle Entries? *	<input checked="" type="checkbox"/>
Generate NLS-enabled prompts and tabs? *	<input checked="" type="checkbox"/>
Generator Default Locale *	en
Generator Locales	nl
Read User Locale From *	Browser Setting
Generate Locale Switcher? *	<input type="checkbox"/>

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:

Read User Locale From *	<code>#{jhsUser.locale}</code>
--------------------------------	--------------------------------

When you check the checkbox **Generate Locale Switcher**, a drop down list will be generated in the global menu 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.

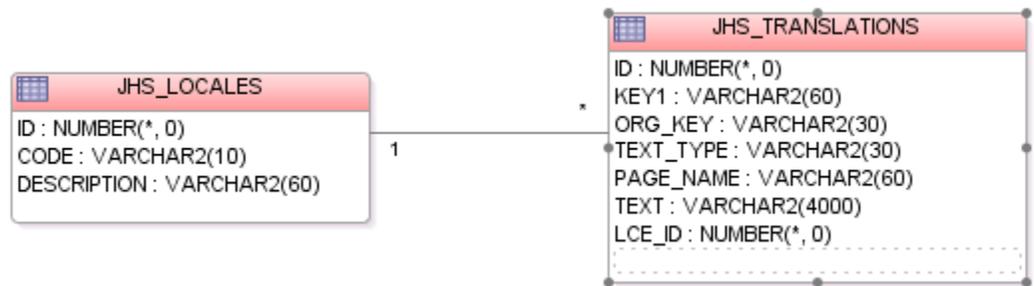
8. Make sure that your application users set the same locale in the browser and in their operating system (Windows: Control Panel – Regional Options). Some language dependent features (in particular ADF Faces) use the browser locale, others the Windows locale.



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 a Database Table to Store Translatable Strings

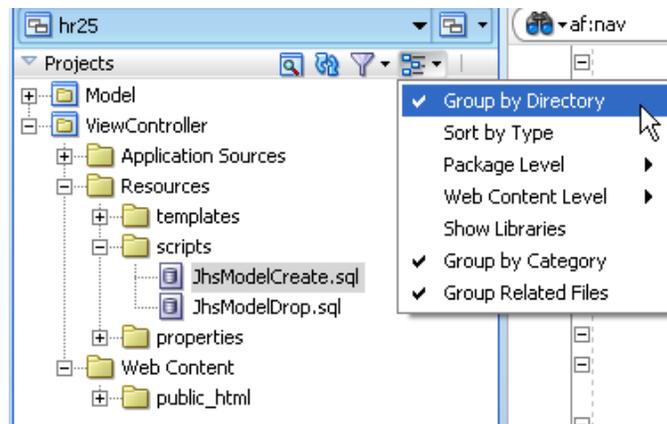
JHeadstart uses two database tables when property **Resource Bundle Type** is set to “databaseTable”. The structure of these tables is shown below.



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 check the “Group By Directory” option in the projects 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.

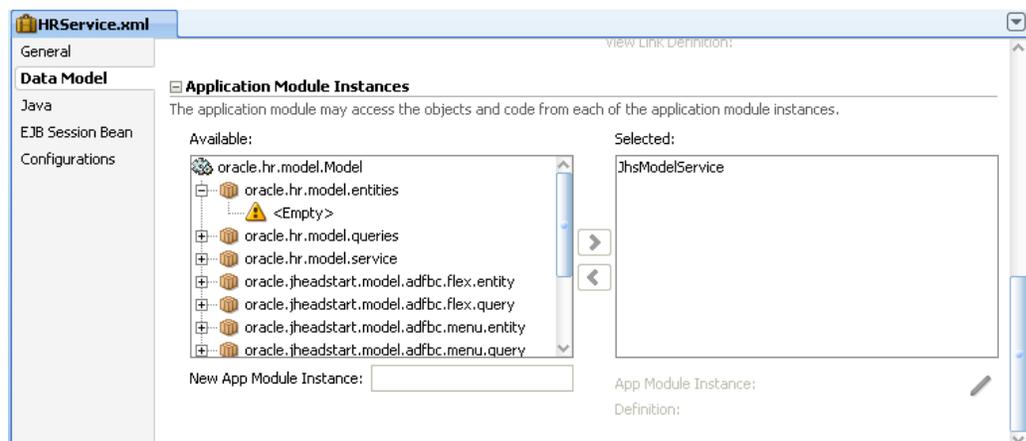
 **Attention:** We recommend installing the JHeadstart tables in the same schema as your own application tables. If you nevertheless prefer to install the JHeadstart tables in a different database schema, then you need to ensure that your application schema has full access to the JHeadstart tables and 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:

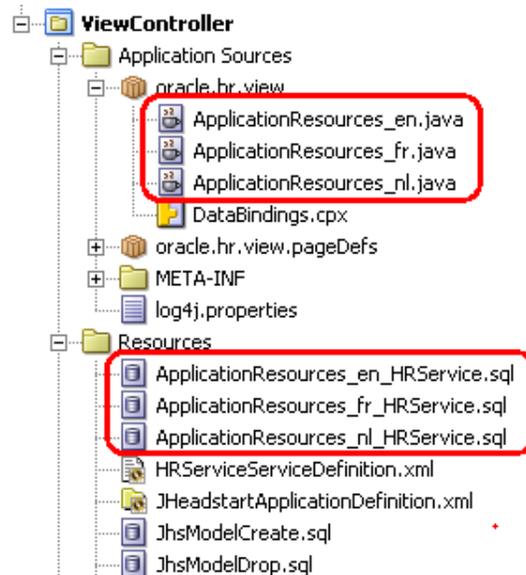
```

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.

- Using the default/misc/file/menuGlobal.vm template, a popup dialog window is generated into the page template 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 links are generated into the global menu area, one link to record the page text, and one link to change/translate the page text.



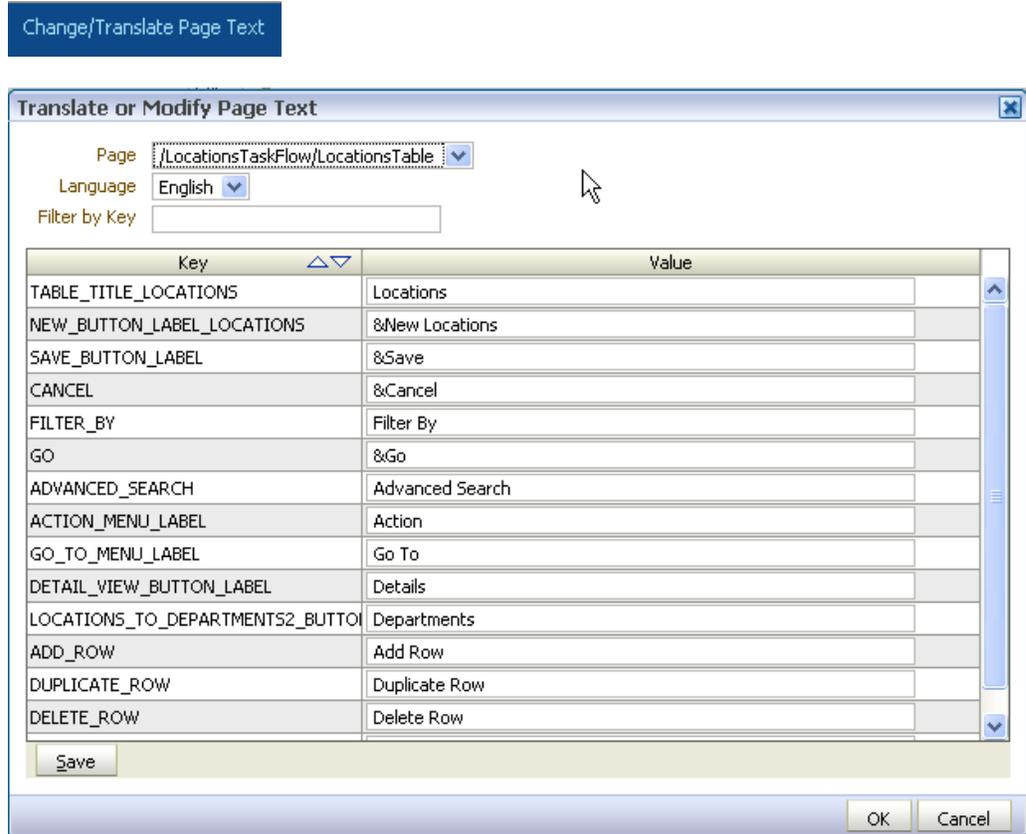
11.2.3. Running the Application

If you now run the generated application, you should see the Record Page Text link in the global menu 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 link, the JHeadstart runtime will “record” all translatable text strings in each page, and the Record Page Text link will be replaced with a link that can be used to invoke the ChangePageText dialog window.



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 set tag to your page like this:

```
<c:set var="storefrontuiBundle"
      value="#{adfBundle['oracle.storefront.ui.StorefrontUIBundle']}" />
```

And then you can access entries in this resource bundle like this:

```
<af:panelHeader title="#{ storefrontuiBundle['TITLE_EMPLOYEES'].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 `<c:set>` tags, and you need to know which entry resides in which bundle.

JHeadstart takes a slightly different approach. Instead of generating `<c:set>` 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:panelHeader title="#{nls['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 messages. To do so, simply include the message key, for example `JHS-00100` in one of your application resource bundles. If the key is not found in your default resource bundle(s), the standard JHeadstart 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

JHeadstart uses a custom error handler class to add functionality to how exception messages are displayed to the user. The custom error handler class is set through the **ErrorHandlerClass** property in **DataBindings.cpx**



The screenshot shows the XML configuration for DataBindings.cpx. The `<Application>` tag includes the `ErrorHandlerClass` property, which is highlighted with a red box. The value is `oracle.jheadstart.controller.jsf.util.JhsDCEErrorHandlerImpl`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Application xmlns="http://xmlns.oracle.com/adfm/application"
  version="11.1.1.51.88" id="DataBindings" SeparateXMLFiles="false"
  Package="oracle.hr.view" ClientType="Generic"
  ErrorHandlerClass="oracle.jheadstart.controller.jsf.util.JhsDCEErrorHandlerImpl">
  <pageMap>
    <page path="/pages/UIShell.jspx" usageId="UIShellPageDef"/>
  </pageMap>
</Application>
```

This class adds the following behavior the default ADF error reporting:

- Some exceptions are skipped and not displayed to the user, because there is a more detailed child exception message
- 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.

If you want to customize or extend this error reporting behavior, you can subclass the `JhsDCEErrorHandlerImpl` class and specify your own subclass in the **ErrorHandlerClass** property of **Databindings.cpx**.



Reference: See the Javadoc or source of `JhsDCEErrorHandlerImpl`

11.5. Outstanding Changes Warning

When the user has made changes to any of the data fields on a page, but then “abandons” the current task flow by clicking on another menu tab or global menu link, JHeadstart will show an alert to ask the user how he wants to handle the pending changes.



When the user clicks Cancel, the navigation action is aborted and the user stays in the original page. When the user clicks the Yes button, a Commit operation is executed, and when the Commit is successful, navigation takes place. If the Commit operation fails because of user errors, the user stays on the original page, allowing the user to fix the data entry errors.

This functionality is implemented through an actionListener property specified on the commandNavigationItem within the menus.

```
<af:commandNavigationItem id="Item" partialSubmit="true"
  text="{menuitem.label}"
  actionListener="{pageFlowScope.pendingChangesBean.handle}"
  action="{menuitem.doAction}"
  rendered="{menuitem.rendered}"/>
```

The actionListener calls the `handle` method on the `PendingChangesBean`. Take a look at the javadoc and/or source code of the `PendingChangesBean` class to see the details of this implementation.



Reference: See the javadoc of `oracle.jheadstart.controller.jsf.bean.PendingChangesBean`.

Note that to be able to detect changes made by the user that were not yet posted to the middle tier, the JSF Model Update phase must be executed before we can detect whether there are pending changes. This means that the `commandNavigationItems` cannot have the `immediate` property set to `true`, as that would skip the JSF Model Update phase. This implies that if the user abandons a task flow with invalid data entered, the user will be presented with error messages, and no menu navigation will take place. To be able to navigate away in this situation, the user should first click the cancel button that is generated on every page, which will rollback the pending (invalid) changes.

When using dynamic tabs the outstanding changes alert will not be shown because each dynamic tab has its own transaction scope, and no data is lost when switching tabs.

11.6. Using Online Help

ADF Faces provides support for integration with online help systems. The format in which the online help is stored varies based on the online help system provider.



Web User Interface Developer's Guide for ADF, Section 19.5 Displaying Help for Components

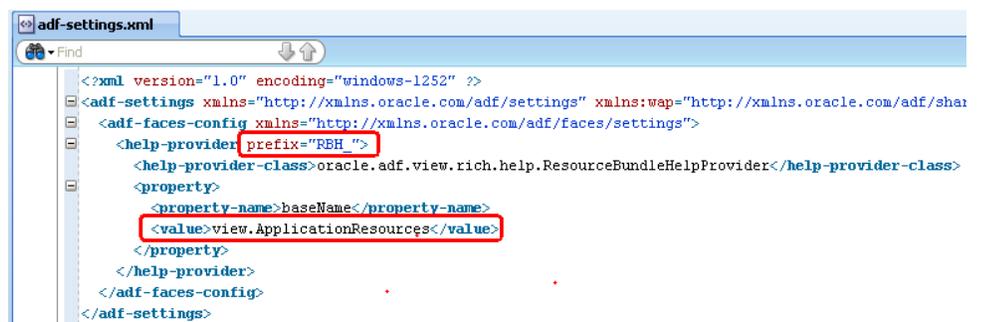
http://docs.oracle.com/cd/E24382_01/web.1112/e16181/af_message.htm#CHDHIGIA

As explained in section 17.5 'Displaying Help for Components' in the Web User Interface Developer's Guide for Oracle ADF, ADF faces supports three types of help provider.

11.6.1. Using the Resource Bundle Online Help Provider

JHeadstart provides most support for help text stored in a resource bundle, as JHeadstart will generate the resource bundle entries for you. Here are the steps to use the resource bundle help provider with JHeadstart:

- Create the `adf-settings.xml` file in the `META-INF` directory of your View Controller project as described explained in section section 17.5 "Displaying Help for Components" in the Web User Interface Developer's Guide for Oracle ADF.



```
<?xml version="1.0" encoding="windows-1252" ?>
<adf-settings xmlns="http://xmlns.oracle.com/adf/settings" xmlns:wap="http://xmlns.oracle.com/adf/sha
<adf-faces-config xmlns="http://xmlns.oracle.com/adf/faces/settings"
<help-provider prefix="RBH_"
<help-provider-class>oracle.adf.view.rich.help.ResourceBundleHelpProvider</help-provider-class>
<property
<property-name>baseName</property-name>
<value>view.ApplicationResources</value>
</property>
</help-provider>
</adf-faces-config>
</adf-settings>
```

- Make sure you set the help provider **prefix** property to "RBH_". This value is used when generating the **helpTopicId** property against the various ADF Faces component in the pages generated by JHeadstart.
- The **baseName** property should be set to the resource bundle generated by JHeadstart. In other words, this property should have the same value as the **NLS Resource Bundle** property that you set at the application level in the JHeadstart Application Definition Editor.

Internationalization	
NLS Resource Bundle *	view.ApplicationResources

- Set the **Online Help Provider** property at application-level to the value shown below.

Online Help Provider	oracle.adf.view.rich.help.ResourceBundleHelpProvider
----------------------	--

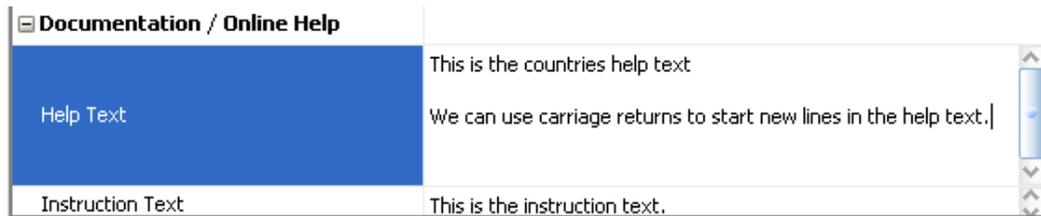
- Use the **Help Text** and **Instruction Text** properties at group level, group, region container, group region, item region and item level to specify online help. See next section for more information.

Note that you can use any value you like for the JHeadstart **Resource Bundle Type** property, they will all work with this online help provider, including `databaseTable`.

If you create a custom subclass for `oracle.adf.view.rich.help.ResourceBundleHelpProvider` for example to supply an external URL to retrieve online help, you need to specify this subclass in `adf-settings.xml`. If you want JHeadstart to still generate resource bundle entries, you should leave the value of the **Online Help Provider** property to the base class.

11.6.2. Using the Help Text and Instruction Text Property

When you specify the **Help Text** property against a group, region container, group region or item region, a help icon will display at the right of the title of this element. In the **Help Text** property you can use carriage returns to start new lines and to create empty lines to divide sections in the help text.



When you specify the **Instruction Text** property against such an element, it will appear below the header line. Carriage returns are ignored, and everything will be displayed on one line, and will wrap to the next line if there is not enough space.



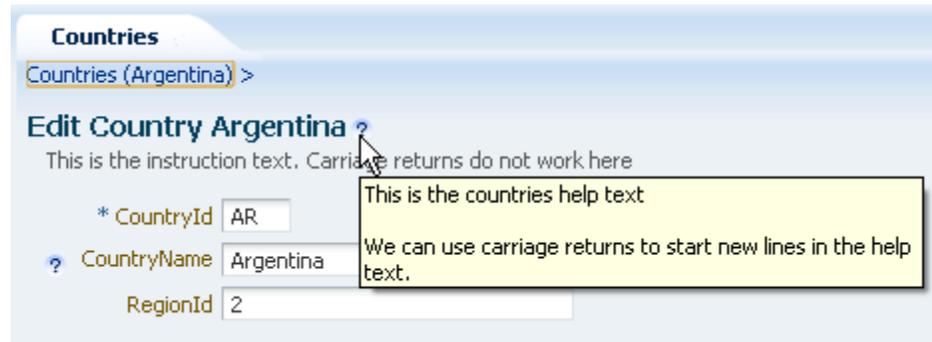
If you hover the mouse over the help icon, the help text appears in a popup window.

Note that both the help icon and the instruction text appear "auto-magically" when the **helpTopicId** property is set on the component.

```
<af:panelHeader id="ph0" helpTopicId="RBH_COUNTRIES"
text="#{pageFlowScope.createModes.Creat
```

ADF Faces will look up two resource bundle entries using the value of the **helpTopicId** property, suffixed with `_DEFINITION` for the help text, and suffixed with `_INSTRUCTION` for the instruction text. If the resource bundle entry does not exist, or does not have a value, the help icon or instruction text will not be displayed.

```
RBH_COUNTRIES_DEFINITION=This is the countries help text'
RBH_COUNTRIES_INSTRUCTIONS=This is the instruction text.'
```



When you specify the **Help Text** property for an item, the help icon will display in front of the item label, as shown above.

The value of the Instruction Text property of an item will appear in a popup window when the focus is in the field. This is similar to the item **Hint (Tooltip)** property.

Hint (Tooltip)	Country name hint text
Depends On Item(s)	
Clear/Refresh Value? *	<input type="checkbox"/>
Operations	
Validation	
Query Settings	
Documentation / Online Help	
Help Text	Country name help text
Instruction Text	Country instruction text.

As a matter of fact, if you specify both the **Hint (Tooltip)** property and the **Instruction Text** property, they will appear in the same popup window, separated by a dotted line.



11.6.3. Using Other Online Help Providers

You can also use another resource bundle provider as documented in section 17.5 "Displaying Help for Components" in the Web User Interface Developer's Guide for Oracle ADF. If you do this, you need to specify the correct class name of this provider in `adf-settings.xml` and in the JHeadstart **Online Help Provider** property. When generating your application with a value for **Online Help Provider** that differs from `oracle.adf.view.rich.help.ResourceBundleHelpProvider`, the **helpTopicId** property is still generated in all components, but you will need to supply the online help resource, like an `.XLIFF` file yourself.



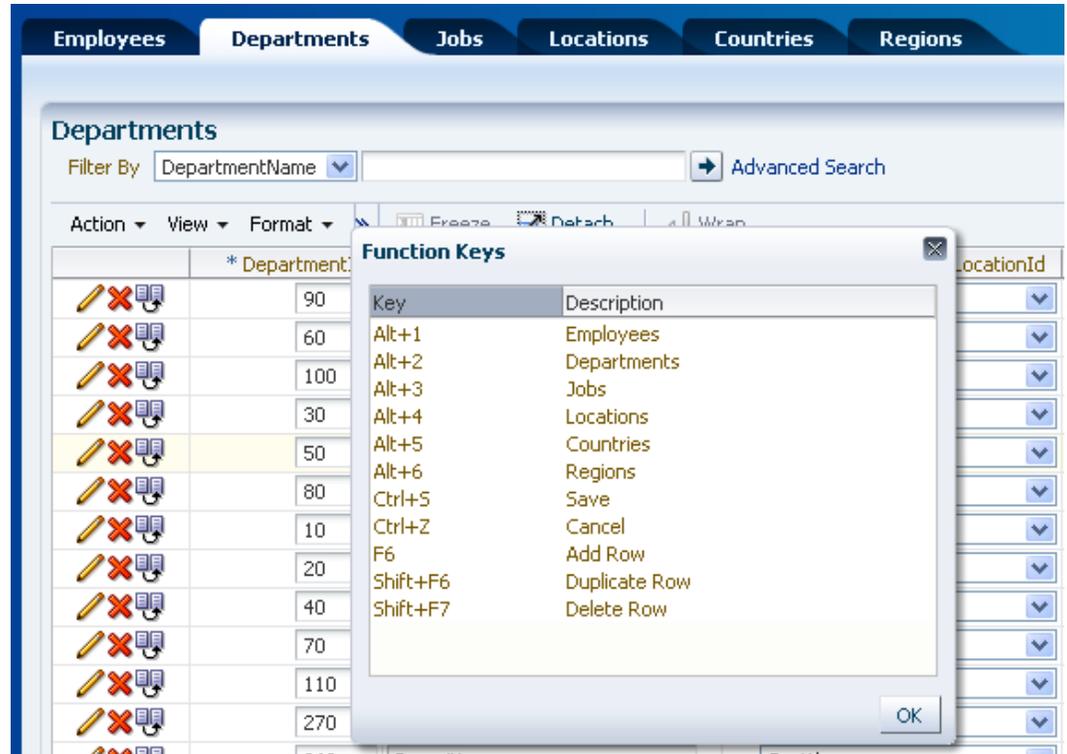
Suggestion: If you still want to record the help and instruction text using the JHeadstart Application Definition Editor, you can create a Velocity template that loops over all groups and items to generate a custom file that produces the required output format, like an `.XLIFF` help file. See chapter 12, section 12.6.3 "How to generate additional custom files".

11.7. Using Function Keys

JHeadstart provides advanced, context-sensitive, support for function keys. All you need to do to enable function keys in your application, is to check the **Enable Function Keys** property that can be found at application level under the **UI Settings** category.



When you generate your application with this setting, you can click Ctrl-K to get a list of function keys. This list is *context-sensitive*, you only see function keys that are available based on the UI component that has focus.



As you can see, you switch menu tabs using the Alt+number key, this also works when using dynamic tabs as discussed in chapter 9 "Creating Menu Structures", section 9.3 "Using Dynamic Tabs when Opening a Menu Item".

By default, the following function keys are supported:

- Ctrl+S: Commit (Save)
- Ctrl+Z: Rollback (Cancel)
- Ctrl+K: ShowKeys
- Ctrl+L: ListOfValues
- F6: CreateRow
- Shift+F6: DuplicateRow
- Shift+F7: DeleteRow

- BACKSPACE: Disable when not in enterable field to prevent navigation to previous page
- Alt+number (1..9) : DoMenuItem

As explained in the next sections, the implementation of the function keys is very flexible, the function key mapping can easily be changed, and new function keys can be defined, that can be mapped to custom buttons.

11.7.1. What Gets Generated When Enabling Function Keys

When you generate the application with function keys enabled, the following artifacts are generated to implement the function key functionality:

- The `jhs11.js` JavaScript library contains a method to register the function keys, and a callback method that queues an event for a server listener
- The UI Shell pages contain an `af:serverListener` to call the managed bean method that handles the function key event.

```
<af:serverListener type="keyboardToServerNotify"
    method="#{viewScope.jhsKeyboardMappingHandler.handleKeyboardEvent}"/>
```

- Additional managed beans are generated into `JhsCommon-beans.xml`: The `jhsKeyboardMappingHandler` managed bean contains a method to register the keyboard mapping. This method is called by the `JhsPhaseListener` which is configured as JSF phase listener in `faces-config.xml` to ensure the function keys are registered for each page. This managed bean also contains a method to handle the registered function keys. This method delegates the implementation of the function key to the `handleKeyCode` method of the `jhsFunctionKeyHandler` managed bean. This managed bean has a list property which holds a set of supported function keys, for each supported function key an instance of the `FunctionKeyBean` is defined in `JhsCommon-beans.xml`, and added to this list property. Each `FunctionKeyBean` instance has managed properties that define the actual keyboard code for the function key, the display code and label that should be used to describe the function key, and the *logical action* that is associated with the function key.
- In each generated page, the group layout container has a set of `f:attribute` tags that map the *logical action* to the ID of the button, or popup, that also performs this action.

```
<!-- DEBUG-BEGIN:GROUP_CONTENT : default/pageComponent/groupCon
<af:panelGroupLayout layout="vertical" id="CountriesTopLc"
    partialTriggers=" CountriesDeleteDialog Co
    <f:attribute name="TopGroupContainer" value="true"/>
    <f:attribute name="DeleteRow" value="CountriesDeletePopup"/>
    <f:attribute name="CreateRow" value="CountriesNewButton"/>
    <f:attribute name="Commit" value="CountriesSaveButton"/>
    <f:attribute name="Rollback" value="CountriesCancelButton"/>
```

11.7.2. What Happens When You Press a Function Key

When you press a registered function key, the callback method in the `Jhs11.js` JavaScript Library queues an event for the `af:serverListener` which eventually leads to a call of the `FunctionKeyHandlerImpl.handleFunctionKey` method.



Using JavaScript in ADF Faces Rich Client Applications. The JHeadstart implementation of function keys is inspired by the techniques discussed in this white paper.
<http://www.oracle.com/technetwork/developer-tools/jdev/1-2011-javascript-302460.pdf>

In this method, the keyboard code of the function key is passed in as argument, which is used to lookup the corresponding `FunctionKeyBean`. The `FunctionKeyBean` contains the *logical action* associated with this function key. The `clientId` of the UI component in focus is also passed into this method as an argument. This `clientId` is used to traverse up the UI component tree of the page to find a group layout container that has an attribute by the same name as the *logical action* of the function key. If this attribute is found, the value of this attribute is used to find the command component that performs the same action as the function key. When the command component is found, a new `ActionEvent` for this component is queued, which causes the same method to be executed as when the user pressed the button.

If you have a background in Oracle Forms, you could see this implementation as the reverse of the `DO_KEY` built-in. In Oracle Forms the functionality of a button can easily be implemented by executing the logic behind a `KEY` trigger using the `DO_KEY` built-in. Well, the JHeadstart implementation can be seen as "`DO_BUTTON`": we lookup the command component that performs the same action as the function key, and queue an event as if the end user pressed the button.

The advantages of this implementation include:

- No duplication of code
- The action performed is guaranteed to be identical to the action executed when the end user presses the associated button.
- When you as a developer modified the default behavior of a standard JHeadstart-generated button by using a subclass in combination with a custom template, then the associated function key will automatically execute the customized method.
- UI components that "listen" to the button through the **`partialTriggers`** property are also automatically refreshed because after the `ActionEvent` has been queued, the method `AdfFacesContext.partialUpdateNotify` is called for the command component.

Note that when the component found through the value of the `f:attribute` is a popup rather than command component, which is the case with the `DeleteRow` action, the popup is shown programmatically.

11.7.3. Customizing the Function Key Implementation

All you need to do to modify the mapping of function keys is creating a custom template for the `REGISTER_FUNCTION_KEY` template. The default implementation of this template (`default/misc/file/registerFunctionKeys.vm`) makes a call to `#{JHS.registerFunctionKey}` for each supported function key.

```

    §{JHS.registerFunctionKey("F6","F6","#ADD_ROW_BUTTON_LABEL()","CreateRow",false)}
    §{JHS.registerFunctionKey("shift F6","Shift+F6","#DUPLICATE_ROW_BUTTON_LABEL()","D
    §{JHS.registerFunctionKey("shift F7","Shift+F7","#DELETE_ROW_BUTTON_LABEL()","Dele
    §{JHS.registerFunctionKey("ctrl S","Ctrl+S","#SAVE_LABEL()","Commit",true)}
    §{JHS.registerFunctionKey("ctrl Z","Ctrl+Z","#CANCEL_LABEL()","Rollback",true)}
    §{JHS.registerFunctionKey("ctrl K","Ctrl+K","#SHOW_KEYS_LABEL()","ShowKeys",true)}
    §{JHS.registerFunctionKey("ctrl L","Ctrl+L","#LIST_OF_VALUES_LABEL()","ListOfValue
    §{JHS.registerFunctionKey("BACK_SPACE","Backspace","Disable Backspace button to pr
    §{JHS.registerFunctionKey("alt 1","Alt+1","Menu Item 1","DoMenuItem",true)}
    §{JHS.registerFunctionKey("alt 2","Alt+2","Menu Item 2","DoMenuItem",true)}
    §{JHS.registerFunctionKey("alt 3","Alt+3","Menu Item 3","DoMenuItem",true)}
    §{JHS.registerFunctionKey("alt 4","Alt+4","Menu Item 4","DoMenuItem",true)}
    §{JHS.registerFunctionKey("alt 5","Alt+5","Menu Item 5","DoMenuItem",true)}
    §{JHS.registerFunctionKey("alt 6","Alt+6","Menu Item 6","DoMenuItem",true)}
    §{JHS.registerFunctionKey("alt 7","Alt+7","Menu Item 7","DoMenuItem",true)}
    §{JHS.registerFunctionKey("alt 8","Alt+8","Menu Item 8","DoMenuItem",true)}
    §{JHS.registerFunctionKey("alt 9","Alt+9","Menu Item 9","DoMenuItem",true)}

```

The `registerFunctionKey` method takes the following parameters:

- the keyboard function key code
- the function key display code that should be used in the "Show Keys" popup window
- the function key description/label that should be used in the "Show Keys" popup window
- the logical action that should be associated with the function key. This action is used to look up the `f:attribute` of the group layout container in the page by the same name, as explained above.
- A boolean that indicates whether it is a page-level action (`true`), or group or item level action (`false`). For page-level actions, only the attributes of the top-level group layout container are inspected.

If you want to change the implementation of a function key, rather than the mapping, you can create a subclass of the `FunctionKeyHandlerImpl` class and override method `handleFunctionKey`. To continue to use your subclass when regenerating your application, you create a custom template for the `FUNCTION_KEY_HANDLER_BEAN` template and modify the class name in this custom template.

For more information on creating custom templates, see chapter 12 "Customizing Generator Output".



Attention: If changes in the function key mapping are not picked up, you might need to clear your browser cache, to make sure you are using the latest version of the `jhs11.js` JavaScript library.

11.7.4. Adding a New Function Key

You can easily add new function keys. We will illustrate the steps needed using an example: we want to generate a custom toolbar button to increase the salary of an employee, and use function key `Ctrl+I` to trigger the salary increase.



First create the toolbar button item as described in chapter 6, section 6.10 "Generating a Button Item". Then perform the following steps:

- Create a custom template for the TOOLBAR_BUTTON template, based on the default template, and add the following line:

```
#ADD_GROUP_ACTION({JHS.page.group} "IncreaseSalary" "{JHS.current.item.id}")
```

- Create a custom template for REGISTER_FUNCTION_KEYS template, and add the following line:

```
{JHS.registerFunctionKey("ctrl I","Ctrl+I","Increase Salary","IncreaseSalary",false)}
```

- Generate and run the application, and test the function key.

Note that the value of the second argument of the #ADD_GROUP_ACTION macro call should be identical to the value of the fourth argument of the \${JHS.registerFunctionKey} call. The #ADD_GROUP_ACTION macro call ensures that an f:attribute tag is added to the employee group layout container, as shown below:

```
<!-- DEBUG:BEGIN:GROUP_CONTENT : default/pageComponent/groupContent.vm, nesti:
<af:panelGroupLayout layout="vertical" id="EmployeesTopLc"
    partialTriggers=" EmployeesDeleteDialog Employeesfbb Emp
<f:attribute name="TopGroupContainer" value="true"/>
<f:attribute name="DeleteRow" value="EmployeesDeletePopup"/>
<f:attribute name="CreateRow" value="EmployeesNewButton"/>
<f:attribute name="IncreaseSalary" value="EmployeesIncreaseSalaryButton"/>
<f:attribute name="Commit" value="EmployeesSaveButton"/>
<f:attribute name="Rollback" value="EmployeesCancelButton"/>
```

11.7.5. Understanding the Difference Between Function Keys and Access Keys

ADF Faces has built-in support for so-called access keys. An access key always uses the ALT key in combination with a letter. The letter is typically underscored in the label of the component by using the `textAndAccessKey` property or just the `accessKey` property. JHeadstart allows you to generate access keys by adding an ampersand in front of the letter that should be underscored in the **Prompt in Form Layout** or **Prompt in Table Layout** property.



So, in the above example the `IncreaseSalaryButton` logic can be invoked by using the `Alt+I` access key, or the `Ctrl+I` function key. You might wonder why we need function keys if access keys already can do the job. Well, there are a few differences between the two:

- Access keys always have the format `Alt+Letter` which limits the number of meaningful access keys you can have in a page.
- If multiple components have the same access key on a page, the last component "wins", there is no way to alter this behavior.
- Access keys are not context-sensitive, if you have a master-detail page, and both the master and detail support insert, you cannot have the same access key for both `CreateRow` buttons. Function keys are context-sensitive, and based on the input focus, the same `CreateRow` function key will execute on either the master or detail group.
- If the browser natively defines `ALT+letter` keys that conflict with your application access keys, then the browser "wins".
- If you define a function key that conflicts with a native browser function key, it depends on the key and browser who "wins". For example, `Ctrl+N` always opens a new browser window in Google Chrome, and will ignore your custom ADF function key. In Internet Explorer, your custom `Ctrl+N` function key wins.

In short, access keys are sometimes too restrictive, in such situations you can use function keys to enable your end users to only use the keyboard to use the application.



Suggestion: Consult the documentation of the browsers you need to support to prevent possible conflicts with function keys and access keys. If you do choose to use a native browser function key then test whether your custom function key "wins".

Customizing Generator Output

This chapter discusses how you can customize the different types of files generated by the JHeadstart Application Generator. While JHeadstart provides a wealth of declarative settings in the JHeadstart Application Definition Editor to implemented very sophisticated functionalities, you might have requirements that cannot be fully implemented by configuring the service definitions in the JHeadstart Application Definition Editor. This chapter explains how you can customize every aspect of the generated application.

The following topics are discussed:

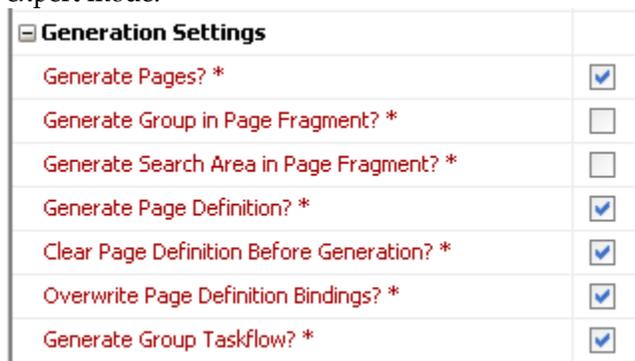
- Recommended approach for customizing JHeadstart generator output.
- Understanding generator architecture and generator templates
- Creating custom templates
- Customizing pages
- Customizing task flows
- Customizing output of the file generator
- Customizing page definitions

12.1. 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 ADF task flow file, and then generate your application again, you would lose 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.



The screenshot shows a table titled "Generation Settings" with the following rows:

Generation Settings	
Generate Pages? *	<input checked="" type="checkbox"/>
Generate Group in Page Fragment? *	<input type="checkbox"/>
Generate Search Area in Page Fragment? *	<input type="checkbox"/>
Generate Page Definition? *	<input checked="" type="checkbox"/>
Clear Page Definition Before Generation? *	<input checked="" type="checkbox"/>
Overwrite Page Definition Bindings? *	<input checked="" type="checkbox"/>
Generate Group Taskflow? *	<input checked="" type="checkbox"/>

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.

- The third option is initially a bit more work and requires you to understand the JHeadstart templating architecture (explained in the remainder of this chapter). 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 adfc-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 you have built an application with JDeveloper/JHeadstart 10.1.3, then regenerating your existing application with JHeadstart 11 will automatically leverage the ADF Faces Rich Components and ADF task flows. 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.

12.2. Understanding Generator Architecture and Generator Templates

This paragraph explains what happens when you run the JHeadstart Application Generator. It explains how generator Java classes and generator templates fit together. Understanding the information in this paragraph will make it much easier for you to write your own custom templates to keep your application 100% generatable.

12.2.1. Velocity and the Velocity Template Language

JHeadstart uses the Velocity template engine to generate output. Velocity is a simple yet powerful Java-based template engine that renders data from plain Java objects to text, xml, email, SQL, Post Script, HTML etc. The template syntax and rendering engine are both easy to understand and quick to learn and implement.

Velocity allows web page designers and other template writers to include markup statements called references in the page. These references are pulled from a Context object that provides get and set methods for retrieving and setting objects. Values retrieved through get methods are inserted directly in a page. Velocity provides basic control statements, that can loop over a collection of values (foreach) or conditionally show a block of text (if/else). The ability to call arbitrary Java methods, include other files, and to create macros that can be repeatedly used make this a powerful yet easy-to-use approach for creating dynamic web page or other text files.

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.



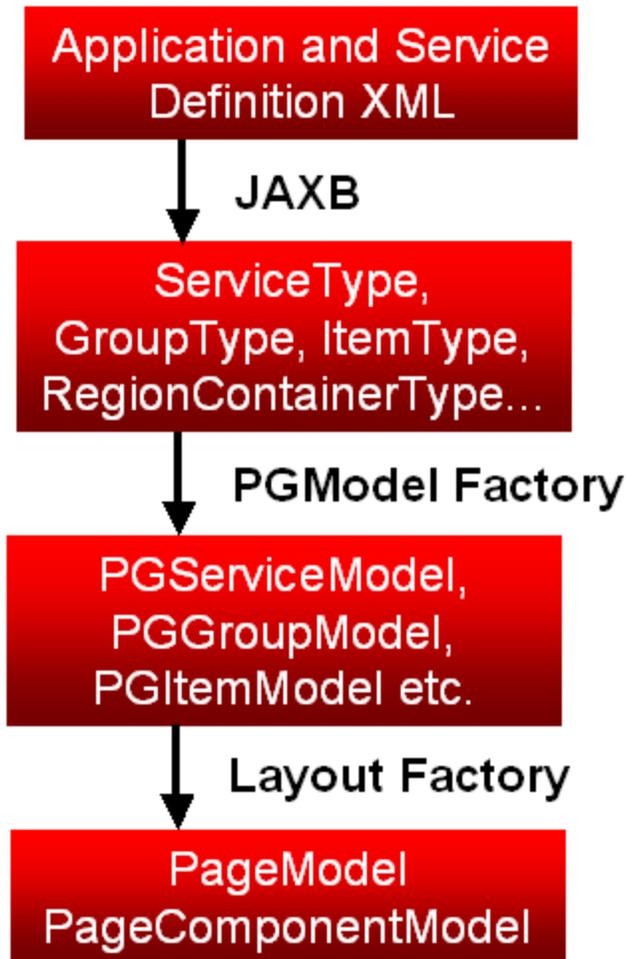
Reference: See the following three documents at the Velocity website:

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

12.2.2. Understanding the JHeadstart Metadata Model

Before we start explaining the JHS-specific constructs used in the various generator templates, it is important to understand a bit more about the generator architecture. When you start generating an application, JHeadstart first constructs a so-called metadata model. This is a set of Java classes that is referenced in the velocity templates.

The picture below explains the different layers in the metadata model.



The first layer consists of the XML files you create through the JHeadstart Application Definition editor. These definitions are stored in XML files, and the structure of these XML files is defined in an XML Schema Definition (XSD) file. You can find this file in `JDEV_HOME\jdev\extensions\oracle.jheadstart.11.1.1\doc\Application Definition.xsd`.

JHeadstart uses JAXB (Java Architecture for XML binding) to create a set of Java classes that provide a 1:1 representation of the information stored in these XML files.

After the JAXB classes are constructed, JHeadstart uses the `PGModelFactory` class to create so-called `PGModel` classes for each object type defined in the application and service definition XML files. These classes wrap the corresponding JAXB class and contain many convenience methods that are used in the velocity templates. By creating these convenience methods in Java the content of the templates remains easier. Complex algorithms to derive information that is to be used in velocity templates are easier to code in Java than in the velocity templates.

For example, the `PGGroupModel` class contains a method `getAdvancedSearchItems()` which can be used in a template to loop over all items that should appear in the advanced search area.

The lowest layer consist of two classes: `PageModel` and `PageComponentModel`, where the first class is a subclass of the second class. JHeadstart uses the `ApplicationLayoutGenerator` class, and various factory classes to create instances

of `PageModel` and `PageComponent` classes. For each page that needs to be generated, one `PageModel` instance is created for the first group shown on the page, and additional `PageComponentModel` instances are created for any detail groups that should appear on the same page. For example, if we have a top-level group `Departments` with **Layout Style** "table-form", and a detail group `Employees` with **Layout Style** "table", with the **Same Page** checkbox checked, the following instances are created:

- `PageModel` instance with layout style "table" for `Departments` group
- `PageModel` instance with layout style "form" for `Departments` group.
- `PageComponentModel` instance with layout style "table" for `Employees` group.

The `PageComponentModel` superclass has a method `getPageComponents()` to retrieve the detail page components, which in the case of the second `PageModel` class will return the `PageComponentModel` instance for the `Employees` group.

Note that in addition to the metadata model classes, there are other model classes used by specific generators. These model classes are not directly derived from the XML metadata, instances of these model classes are created when running specific generators. See section [Understanding Application Generators](#) in this chapter for more information.

The javadoc of all these classes can be found through the **JHeadstart Documentation Index**, available from the JDeveloper Help menu.

Oracle JHeadstart 11.1.1 - Documentation Index

Release 11.1.1.3.23

Included with JHeadstart

- [Release Notes](#)
- [Javadoc of the JHeadstart Runtime ADF Extensions](#)
- [Javadoc of the JHeadstart Application Generator](#)
- [Javadoc of the JHeadstart Metadata Model JAXB Classes](#)

This javadoc can be helpful to see all the Java methods available to retrieve information that you can use in a velocity template, as we will see below.

12.2.3. Referencing JHeadstart Metadata Model in Velocity Templates

The top-level context object as required by Velocity is an instance of the `JhsVelocityContext` class, and is available through the key "JHS". This is why all JHeadstart-specific constructs in generator templates start with `{ JHS`

To access the various `PGModel` instances, you can use constructs like this:

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

- etc.

While these constructs return instances of the `PGModel` classes, and not the wrapped JAXB classes, you can directly access the methods of the JAXB classes from here because a fallback mechanism has been implemented that looks up the JAXB instance if a requested getter method is not found on the `PGModel` class. This means that to get an overview of all methods you can call on the current group, you should look at the Javadoc of both the `PGGroupModel` class, as well as the `JAXB GroupType` class.

For example, you can use `${JHS.current.group.helpText}` to get the group help text. The `getHelpText()` method is not implemented on `PGGroupModel` but is available on the `JAXB GroupType` class.

The `PGGroupModel` class itself has a convenience method `getParentGroup()`, so you can use `${JHS.current.group.parentGroup.name}` to get the name of the parent group.

Note that to call a getter method without arguments, you can use Java bean-like property notation, omitting the "get" prefix and brackets. You can also call any Java method directly by specifying the complete Java method name, including any arguments. So, to get the help text, you can also use the following notation:
`${JHS.current.group.getHelpText() }`

If the Java method you want to call has arguments, you can pass in these arguments within the brackets, just like coding in Java:

```
${JHS.stripBrackets (${group.updateAllowedExpression}) }
```

To access the `PageModel` and `PageComponentModel` instances, you can use:

- `${JHS.page}`
- `${JHS.current.pageComponent}`

Note that the generator always runs into the context of only one application, one service, and one page (when generating pages), this is why these instances have references without the intermediate `.current.` construct.

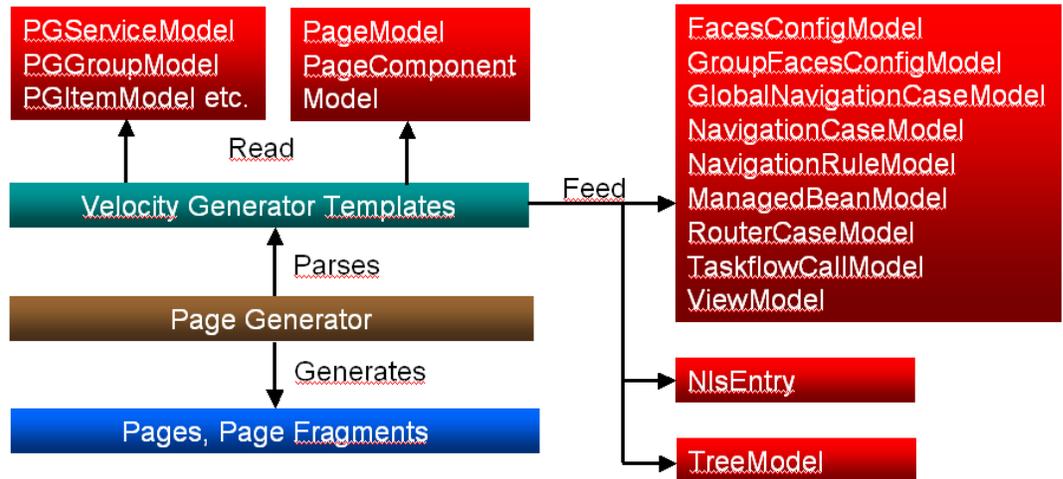
12.2.4. Understanding Application Generators

When you generate an application, and the metadata model as discussed above has been constructed, various generator classes are used to produce the different file types, as shown in the table below.

Generator	Velocity Template Access	Output
Page (Fragment) Generator	<code>\${JHS.pageGenerator}</code>	.jsf and .jsff files
AdfcConfig (Task flow) Generator	<code>\${JHS.facesConfigGenerator}</code>	Adfc-config XML files
Menu Generator	<code>\${JHS.menuGenerator}</code>	XMLMenuModel XML files
NLS Generator	<code>\${JHS.nlsGenerator}</code>	Resource bundles, or SQL scripts

Tree Generator	<code>#{JHS.treeGenerator}</code>	Tree .jsff files
PageDefinition generator	<code>#{JHS.pageDefGenerator}</code>	PageDefinition XML files
File Generator	<code>#{JHS.fileGenerator}</code>	Miscellaneous files

The sequence in which the generators run is important. This is because some generators *feed* the metadata model of other generators. This is a very powerful and important concept to understand when you start creating your own custom templates. The picture below shows how the Page (Fragment) Generator *reads* metadata models and *feeds* other generator model classes.



Attention: The generator class and method names still use the standard JSF naming, as generated in JDeveloper 10.1.3. JHeadstart 11 now generates ADF task flows, with control flow rules and control flow cases instead of navigation rules and navigation cases, however the class and method names have not been changed.

A snippet from the "Details" button template that navigates from the Select page to the Form page will clarify the above picture:

```
<af:commandButton
    textAndAccessKey="#EDIT_BUTTON_LABEL()"
    action="#{JHS.facesConfigGenerator.addNavigationCase(#{JHS.current.group},#{JHS.page.name}
        , "details", #{JHS.current.pageComponent.detailsPage.name)}"
```

The `textAndAccessKey` property references a reusable macro `#EDIT_BUTTON_LABEL`, which looks like this:

```
#macro (EDIT_BUTTON_LABEL)
#{JHS.nls(#{JHS.current.group.displayTitleSingular, "EDIT_BUTTON_LABEL_#{JHS.current.group.name}"
    , "EDIT_BUTTON_LABEL")}#end
```

When this template is parsed, the `#{JHS.nls}` call creates a new instance of the `NlsEntry` model class, which is later picked up by the NLS Generator, to generate the following entry in the resource bundle:



Likewise, the action property contains a method call to `#{JHS.facesConfigGenerator.addNavigationCase}` which will create a new instance of the `NavigationCaseModel` class, which is later picked up by the `AdfcConfig` generator to generate the following control-flow-case:

```
<control-flow-rule id="_133">
  <from-activity-id id="_134">SelectRequestStatuses</from-activity-id>
  <control-flow-case id="_135">
    <from-outcome id="_136">details</from-outcome>
    <to-activity-id id="_137">RequestStatuses</to-activity-id>
  </control-flow-case>
  <control-flow-case id="_138">
    <from-outcome id="_139">CreateRequestStatuses</from-outcome>
    <to-activity-id id="_140">RequestStatuses</to-activity-id>
  </control-flow-case>
</control-flow-rule>
```

The XML snippet shows a control-flow-rule with two control-flow-cases. The first case, with id "_135", is highlighted with a red box. It has a from-outcome of "details" and a to-activity-id of "RequestStatuses".

So, by feeding other generator models while generating a page, we ensure that any translatable string on the page ends up in the resource bundle, and that any button that should result in page navigation has a corresponding control flow rule and control flow case. You can use this same feeding mechanism in your custom templates, as is explained in more detail in the section [Creating Custom Templates](#).

12.2.5. Configuration of Generator Classes

Under the covers, JHeadstart uses the Spring bean factory to instantiate the various generator classes. The configuration file that defines the bean classes is `jag-config.xml`, located in the `templates/config` directory of your `ViewController` project. You typically do not need to modify this file, but you might want to take a look at the content to further increase your understanding of the generator architecture.

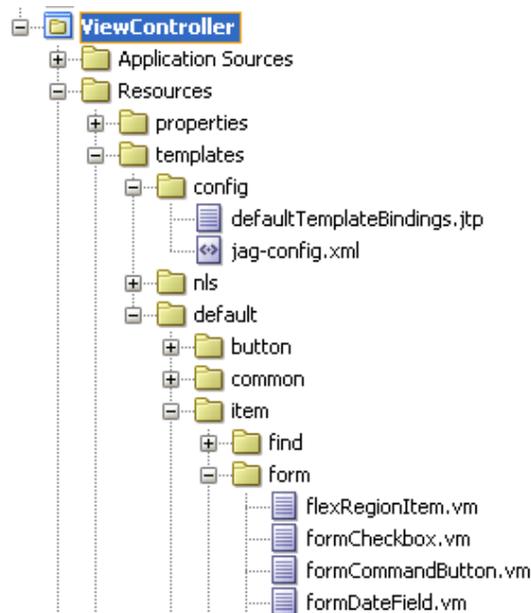
Below, you can see a snippet of this file that instantiates some generator classes. Notice the properties that specify the root templates that should be used to generate specific files.

```
<bean name="menuGenerator" class="oracle.jheadstart.dt.jag.menugenerator.MenuGenerator" singleton="false">
  <property name="xmlMenuModelTemplateIdentifier"><value>XML_MENU_MODEL</value></property>
  <property name="xmlRootMenuModelTemplateIdentifier"><value>XML_ROOT_MENU_MODEL</value></property>
</bean>
<bean name="nlsGenerator" class="oracle.jheadstart.dt.jag.nlsgenerator.NlsManager" singleton="false">
  <property name="viewTypesManager"><ref bean="viewTypesManager"/></property>
  <property name="messageSource"><ref bean="generatorMessageSource"/></property>
  <property name="propertyResourceBundleTemplateIdentifier"><value>NLS_PROPERTY_RESOURCEBUNDLE</value></property>
  <property name="databaseResourceBundleTemplateIdentifier"><value>NLS_DATABASE_RESOURCEBUNDLE</value></property>
  <property name="databaseResourceBundleSqlTemplateIdentifier"><value>NLS_DATABASE_RESOURCEBUNDLE_SQL</value></property>
  <property name="javaResourceBundleTemplateIdentifier"><value>NLS_JAVA_RESOURCEBUNDLE</value></property>
</bean>
```

12.2.6. Structure of Generator Templates

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, as explained above.
- **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



Templates are referenced by logical names. The logical name is mapped to a physical template file in the `defaultTemplateBindings.jtp` file. Below you see a small part of the content of this file:

```
GROUP_CONTENT=default/pageComponent/groupContent.vm
GROUP_TOOLBAR=default/pageComponent/groupToolbar.vm
BREADCRUMB_AREA=default/pageComponent/breadcrumbArea.vm
FORM_GROUP=default/pageComponent/formGroup.vm
```

JHeadstart uses a very-fine grained, and deeply nested template structure, as is partly shown in the structure above. When a file is generated, the generator starts with a root template that is specified in `jag-config.xml`.

```
<bean name="formPage" class="oracle.jheadstart.dt.jag.applicationlayoutgenerator.PageModel"
  <property name="nameFormat"><ref bean="formPageNameFormat" /></property>
  <property name="pageTemplateIdentifier"><value>DATA_PAGE</value></property>
  <property name="regionTemplateIdentifier"><value>FRAGMENT_PAGE</value></property>
  <property name="contentTemplateIdentifier"><value>FORM_PAGE_CONTENT</value></property>
  <property name="groupContentTemplateIdentifier"><value>FORM_GROUP</value></property>
```

The root template will then parse a series of nested templates using the `#JHS_PARSE` macro. The screen shot of the page shown on the next page used 15 templates during generation.

Edit Request Status New request [1 / 7] Save Cancel

* Id

* Code

Description

This is the nested structure of logical template names used for this page:

```

FRAGMENT_PAGE
  FORM_PAGE_CONTENT
    PAGE_CONTENT
      GROUP_CONTENT
        GROUP_TOOLBAR
          FORM_BROWSE_BUTTONS
          NEW_BUTTON
          DELETE_BUTTON
            DELETE_WARNING_DIALOG
          SAVE_BUTTON
          CANCEL_BUTTON
        FORM_GROUP
          FORM_TEXT_INPUT
          FORM_TEXT_INPUT
          FORM_TEXT_INPUT

```

And here is a snippet of the generated page:

```

<af:pageTemplate id="pt" viewId="/common/pageTemplates/JhsRegionTemplate.jspx">
  <!-- DEBUG:BEGIN:FORM_PAGE_CONTENT : default/page/formPageContent.vm, nesting
  <!-- DEBUG:BEGIN:PAGE_CONTENT : default/page/pageContent.vm, nesting level: 3
  <f:facet name="pageContent">
    <!-- Pushed "disabled" on stretchContext, now: Stretching disabled on group
    <af:panelGroupLayout id="pcpg" layout="scroll">
      <!-- DEBUG:BEGIN:GROUP_CONTENT : default/pageComponent/groupContent.vm, ne
      <af:panelGroupLayout layout="vertical" id="RequestStatusesTopLc"
        partialTriggers=" RequestStatusesfbb RequestStatusesf
      <f:facet name="separator">
        <af:spacer width="10" height="10" id="sp0"/>
      </f:facet>
      <!-- Pushed "disabled" on stretchContext, now: Stretching disabled on gr
      <af:panelHeader id="ph0"
        text="#{pageFlowScope.createModes.CreateRequestStatuses
      <f:facet name="toolbar">
        <!-- DEBUG:BEGIN:GROUP_TOOLBAR : default/pageComponent/groupToolbar.
        <af:toolbox id="tbox0">
          <af:toolbar id="tbar0">
            <!-- DEBUG:BEGIN:FORM_BROWSE_BUTTONS : default/pageComponent/for
            <af:panelGroupLayout id="RequestStatusesfbb" layout="horizontal'

```

As you can see, it is quite easy to identify which part of the page is generated by which template, this is added as comments onto the page. This template source information greatly eases the creation of custom templates as you will see later on.

12.3. Creating Custom Templates

We will start with a word of warning: **never** customize one of the default JHeadstart template files directly. When you upgrade to a newer version of JHeadstart they will be overwritten. Always create a custom template, you can create custom templates from scratch, or base them on a default template, as explained below.

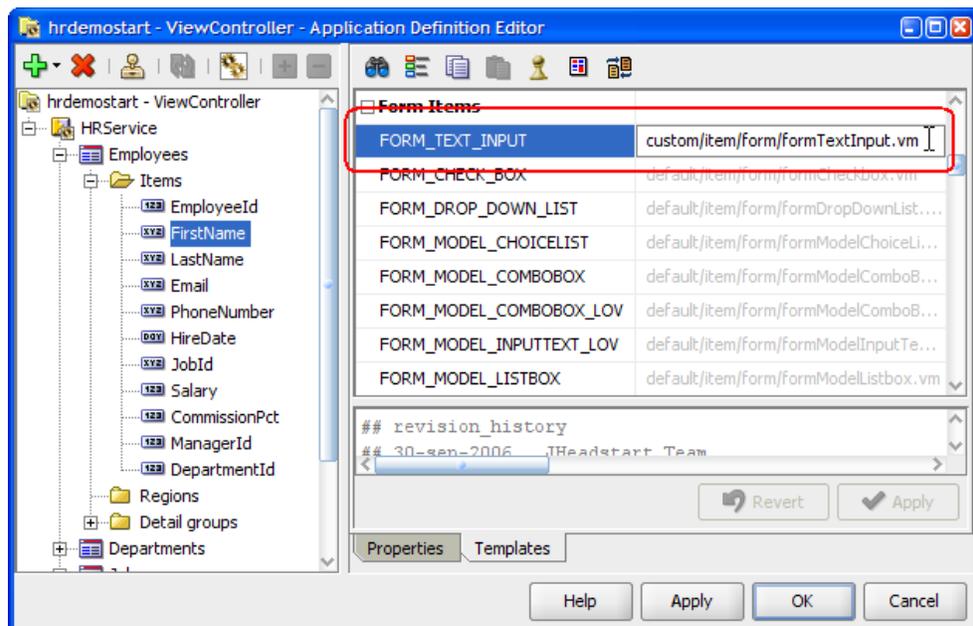
12.3.1. Creating a Custom Template File

To create a custom template, you click on the **Templates** tab in the JHeadstart Application Definition editor. The list of templates you will see is context-sensitive, it is determined by the node selected in the navigator pane at the left side. For example, if you click on the top-level application node, all templates will be shown under the **Templates** tab. If an item is selected in the navigator, only item-level templates are shown.

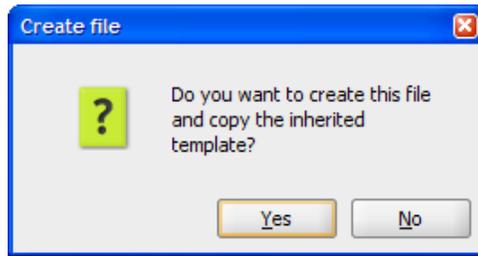
When creating a custom template, the node selected in the navigator also determines the **scope to which the custom template will apply**. For example, when creating a custom item for a date field in form layout (FORM_DATE_FIELD), and the application node is selected, this custom template will be applied to *all* date items displayed in form layout in the entire application. If one specific date item is selected in the navigator pane, the custom template will only be applied to this one item.

Once you have identified the logical template name for which you want to create a custom template, you click in the field with the grey velocity template filename; it will become editable.

Type the name of the (relative) path to the custom template that you wish to use. Best practice is to start the path with 'custom/' (or your application or company name) instead of 'default/'; so there will be a clear separation between the default JHeadstart templates, and your own custom templates.



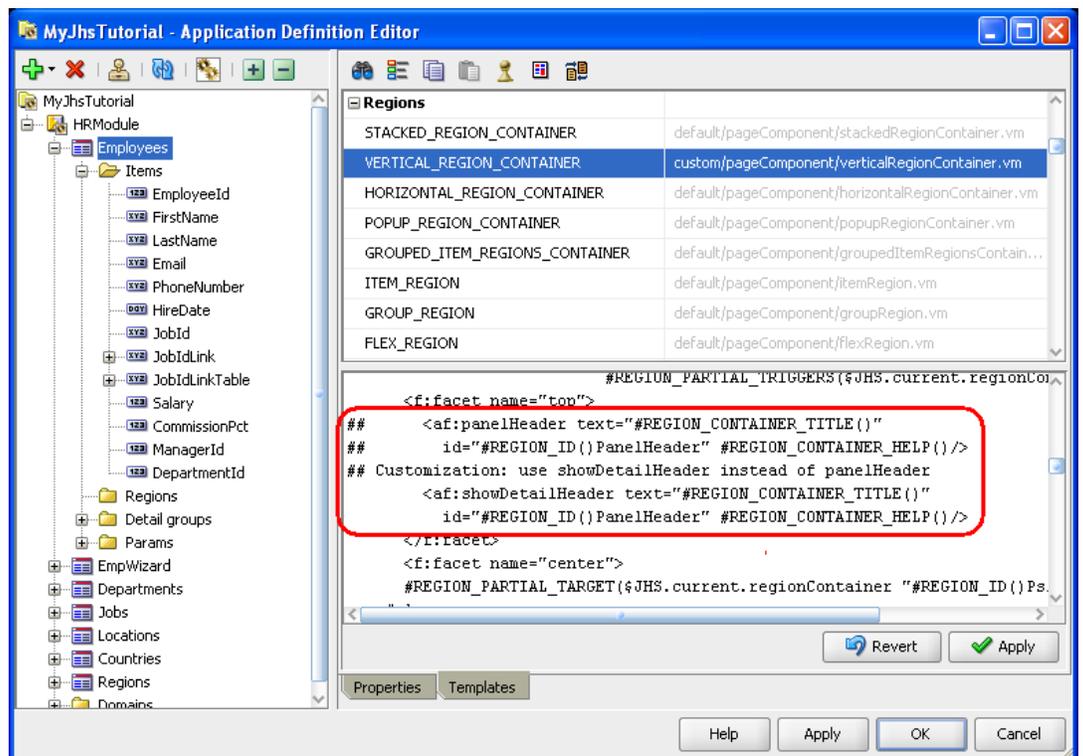
When you have entered the new filename, press the **Enter** key to confirm the new location. A dialog will appear:



Click 'Yes' to copy the template to the location you specified.

The Application Definition Editor now also gives you the option to edit the template itself (it does **not** give you this option for default templates, since you are not supposed to edit them!).

You can freely type your changes in the window, such as the extra attribute in the example below. Click Apply to save the changes to the template.



If you do not want to base the custom template on the content of the default template, you simply clear all content before adding your own template code.



Suggestion: We recommend that you document in the custom template which changes you made compared to the default template. When a new JHeadstart version is released, the corresponding default template might be changed, and you need to reconcile the changes in the default template with your custom template. This is easier when you clearly documented your changes. You can use the "##" characters to start a comment line in Velocity as shown above.

12.3.2. 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:

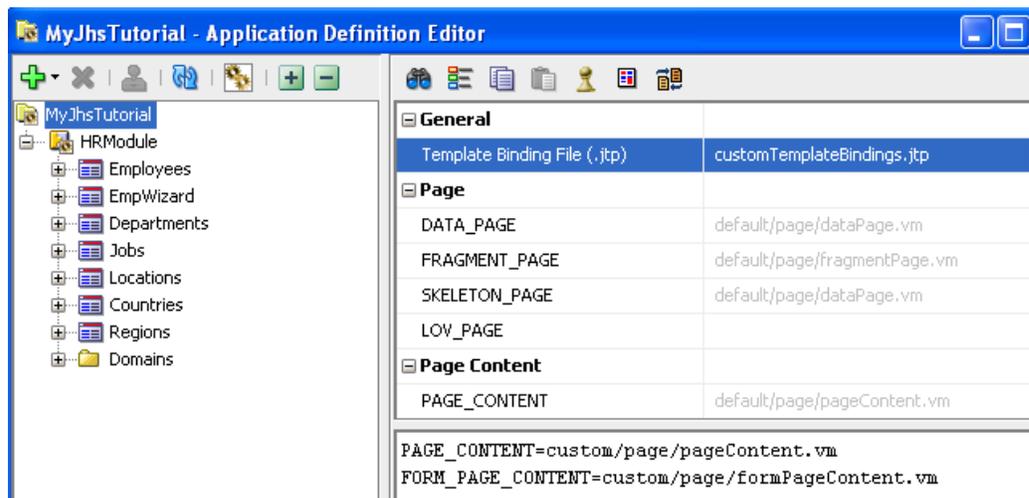
```
<!-- DEBUG:BEGIN:TABLE_TEXT_INPUT : default/item/table/tableTextInput.vm -->
<af:column sortable="true" noWrap="true"
           sortProperty="FirstName">
  <f:facet name="header">
    <af:outputLabel value="FirstName"
                  styleClass="af_column_header-text"/>
  </f:facet>
  <af:inputText id="EmployeesFirstName" value="#{row.FirstName}"
               required="#{bindings.EmployeesFirstName.mandatory}"
               rows="#{bindings.EmployeesFirstName.displayHeight}"
               columns="#{bindings.EmployeesFirstName.displayWidth}"
               maxLength="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, the `TABLE_TEXT_INPUT` template is used which maps to the default template `default/item/table/tableTextInput.vm`.

12.3.3. Grouping Custom Templates

If you have multiple custom templates that should be used together to implement some functionality, you can group these templates in a custom template bindings file. You only need to specify this custom template binding file to have all custom templates used together. Again, the node selected in the navigator pane determines the scope to which the set of custom templates apply.

You can define a custom template binding file to be used for the whole application, a service, or for a group, or for any other level. You specify it by going to the **Templates** and setting the **Template Binding File** property, always shown at the top, to the name of your custom template bindings file.



 **Attention:** A custom template bindings file must always be placed in the templates/config directory of your application, in the same directory as the defaultTemplateBindings.jtp file.

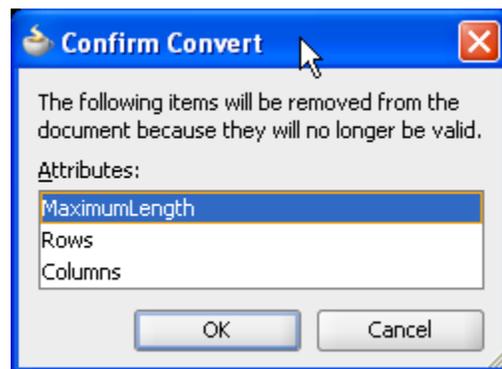
 **Attention:** The Template Binding File you specify in your Application Definition does not need to include the complete list of templates. Only include the lines for the templates you have customized. The other templates will be inherited from the higher level Template Bindings file, or if there is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

12.3.4. Writing Reusable Custom Templates

As explained in the first paragraph of this chapter [Recommended Approach for Customizing JHeadstart Generator Output](#), we recommend to use the JDeveloper visual design time tools to add functionality you cannot generate out-of-the-box, and then move the changes you made to a custom template. In this section, we will go through these steps, and will discuss how to make your custom template code more generic so it can be reused in multiple places.

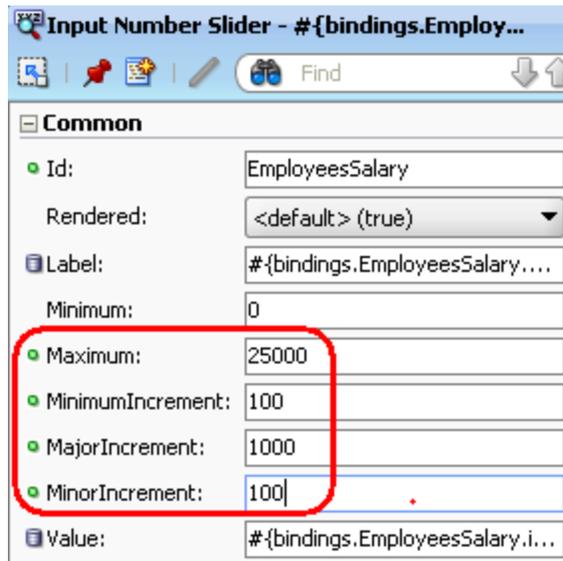
We will use an example to explain the steps to take: we want the employee salary to be updated using a slider component. Here are the steps to implement this:

In the visual design editor, open the generated employee page, right-mouse-click on the Salary item and choose **Convert To..** from the popup menu. Select **Input Number Slider** from the list of elements displayed. The following alert will be shown:



It is good to remember the attributes that are no longer valid, because they should be removed from the reusable custom slider template that we will create later on.

Now, add specific properties for the slider as shown below.



Run the application again to see whether the slider displays and behaves as expected. (Do NOT generate the application again, this would wipe out the changes you just made!)



If you are satisfied with how the slider looks, you go to the source of the employee page, and copy the page snippet that renders the slider.

```
<!-- DEBUG-BEGIN:FORM_TEXT_INPUT : default/item/form/formTextInput.vm, nesting level: 6 -->
<af:inputNumberSlider id="EmployeesSalary" value="#{bindings.EmployeesSalary.inputValue}"
    label="#{bindings.EmployeesSalary.hints.label}"
    required="#{bindings.EmployeesSalary.hints.mandatory}"
    shortDesc="#{bindings.EmployeesSalary.hints.tooltip}" maximum="25000"
    minimumIncrement="100" majorIncrement="1000" minorIncrement="100">
    <f:validator binding="#{bindings.EmployeesSalary.validator}"/>
    <af:convertNumber groupingUsed="false" pattern="#{bindings.EmployeesSalary.format}"/>
</af:inputNumberSlider>
<!-- DEBUG-END:FORM_TEXT_INPUT : default/item/form/formTextInput.vm, nesting level: 6-->
```

Now, the fastest way to keep your page 100% generatable is to create a custom template `employeeSalarySlider.vm` and copy and paste the above page snippet into this template. This way, the template cannot be reused for other salary items in other groups, as it contains hard coded references to the employee item.

In order to make the template more reusable, we need to replace the hard coded references to the employee salary item with dynamic velocity constructs that we can find in the default `formTextInput.vm` template. Here is the sample code for such a custom `salarySlider.vm` template:

```
formSalarySlider.vm
Find
#START_ITEMS_DISPLAYED_AT_RIGHT()
<af:inputNumberSlider #ITEM_ID_IN_FORM() #ITEM_VALUE_IN_FORM() #ITEM_PROMPT_IN_FORM()
  #ITEM_PARTIAL_TRIGGERS() #ITEM_REQUIRED_IN_FORM()
  #ITEM_READ_ONLY_IN_FORM() #ITEM_RENDERED_IN_FORM()
  #ITEM_DISABLED_IN_FORM() #ITEM_HINT() #ITEM_HELP() #ITEM_ADDITIONAL_PROPERTIES()
  #DEPENDS_ON_ITEM_PROPS_FORM() #ITEM_SIMPLE() maximum="25000"
  minimumIncrement="100" majorIncrement="1000" minorIncrement="100">
  #VALIDATOR_BINDING()
  #NUMBER_CONVERTER()
  #ITEM_CONTEXT_FACET()
</af:inputNumberSlider>
#END_ITEMS_DISPLAYED_AT_RIGHT()
```

To get this template, we made the following changes to the default `formTextInput.vm`:

- We changed the ADF Faces element from `af:inputText` to `af:inputNumberSlider`.
- We removed the marco calls to `#ITEM_MAXIMUM_LENGTH()`, `#ITEM_ROWS()` and `#ITEM_COLUMNS()` since these macro's generate properties that are not applicable for a number slider, as we learned when converting the item in the visual page editor.
- We added the four properties specific for the number slides that we set before in the property palette.

Now we can reuse the template for all salary items in our application. But we can make the template even more reusable, we can turn it into a generic number slider template. This means that we need to replace the hard coded values for the four properties specific to the number slider, since these values are specific for salary items.

There are two ways to do this:

- Using the five custom properties available at most of the element types in the JHeadstart Application Definition Editor
- Using the item-specific property **Additional Properties**

12.3.4.1. Using Custom Properties

To be able to set custom properties on an element, you need to switch the JHeadstart Application Definition Editor to expert mode. You can do this by clicking the chess icon in the toolbar () which then changes to a "Queen" icon (.

Now, enter the custom properties as shown below.

Customization Settings	
Additional Properties	
Custom Property 1 Name	Maximum
Custom Property 1 Value	25000
Custom Property 2 Name	MaxIncrement
Custom Property 2 Value	1000
Custom Property 3 Name	MinIncrement
Custom Property 3 Value	100
Custom Property 4 Name	
Custom Property 4 Value	

To get the value of a custom property in your custom template, there are three expressions you can use, for example for group custom properties you can use:

- `${JHS.current.group.property1}`, this is the existing expression that also works when no name is entered for the property
- `${JHS.current.group.getCustomProperty("propertyName")}`, where `propertyName` should be substituted with the name you entered for this property.
- `${JHS.current.group.propertyName}`, where `propertyName` should be substituted with the name you entered for this property. If you use this expression and the group does not have this custom property defined, you will get an error during generation.

We recommend to use the second or third expression, these expressions do not care which custom property number you used to enter the name and value. In addition, if you use the second or third expression, you never run out of custom properties, because you can also enter a comma-delimited list of names and values in the name and value property.

We will use the third expression type in our custom template to reference the value entered for the first custom property. The final, highly reusable `formNumberSlider.vm` template looks like this:

```

numberSlider.vm x
Find
#START_ITEMS_DISPLAYED_AT_RIGHT()
<af:inputNumberSlider #ITEM_ID_IN_FORM() #ITEM_VALUE_IN_FORM()
  #ITEM_PROMPT_IN_FORM() #ITEM_ADDITIONAL_PROPERTIES()
  #ITEM_PARTIAL_TRIGGERS() #ITEM_REQUIRED_IN_FORM()
  #ITEM_READ_ONLY_IN_FORM() #ITEM_RENDERED_IN_FORM()
  #ITEM_DISABLED_IN_FORM() #ITEM_HINT() #ITEM_HELP()
  #DEPENDS_ON_ITEM_PROPS_FORM() #ITEM_SIMPLE()
  maximum="{JHS.current.item.Maximum}"
  majorIncrement="{JHS.current.item.MaxIncrement}"
  minorIncrement="{JHS.current.item.MinIncrement}"
  minimumIncrement="{JHS.current.item.MinIncrement}" >
#VALIDATOR_BINDING()
#ITEM_CONTEXT_FACET()
</af:inputNumberSlider>
#END_ITEMS_DISPLAYED_AT_RIGHT()

```

Note that if you use this generic template with an item that has not all three custom properties filled in, you will get a generation error like this:

```
[Employees.jsff, custom/item/form/numberSlider.vm] Getter for
attribute MinIncrement or custom property MinIncrement not found
in PGItemModel (Salary)
```

If you have a custom template which has optional custom properties, you should use the following statement (we assume MinIncrement property is optional)

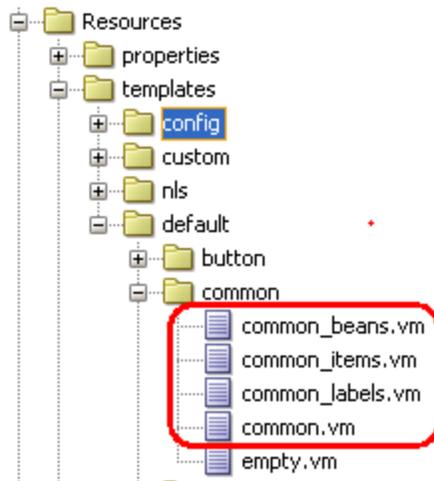
```
#JHS_PROP("minorIncrement"
${JHS.current.item.getCustomProperty("MinIncrement")})
```

As a final note, instead of using three custom properties 1, 2 and 3, you could have defined all three properties using the comma-separated syntax in custom property 1:

Customization Settings	
Additional Properties	
Custom Property 1 Name	Maximum,MaxIncrement,MinIncrement
Custom Property 1 Value	25000,1000,100
Custom Property 2 Name	
Custom Property 2 Value	

12.3.5. Customizing JHeadstart Macro's

The default templates use many reusable macro's that are defined in four Velocity macro files, located in the `templates/default/common` directory.



The use of these macro files is configured in `jag-config.xml` in the `libraries` property of the `velocityInitializer` bean. You can add your own custom macros file *at the end of this property list*, and copy any macro you want to customize to your custom macro file. Because your macro file is added at the end, it takes precedence over the default macro files, and will Velocity will first search the custom macro file for the requested macro.

```

<!-- Velocity initializer -->
<bean name="velocityInitializer"
      class="oracle.jheadstart.dt.jag.engine.velocity.VelocityInitializer"
      singleton="true">
  <property name="libraries">
    <list>
      <value>default/common/common.vm</value>
      <value>default/common/common_items.vm</value>
      <value>default/common/common_labels.vm</value>
      <value>default/common/common_beans.vm</value>
      <value>custom/common/customMacros.vm</value>
    </list>
  </property>

```

And remember the warning in the previous paragraph, you need to re-apply this change to `jag-config.xml` after installing a new JHeadstart version.

12.3.6. Coding and Debugging Tips

Here are some final tips when coding custom Velocity templates:

- The notation `$JHS.current.group` is a shortcut for `${JHS.current.group}`. Always use the extended notation with additional curly brackets when concatenating a Velocity expression with another expression or with literal text.
- When a macro calls another macro in the same library or template, the called macro must be sequenced before the calling macro. Otherwise a confusing error message is shown!
- Velocity commands cannot wrap, they must be coded on a single line. Wrapping causes an error message during generation.

- If you write a Boolean expression, for example in an if-statement, both sides of the expression must have a value (and must be of the same type). If a variable might be null, you should first test whether this variable has a value, as follows:

```
#if ($myOptionalVar)
    #if ("$myRequiredVar"=="$myOptionalVar")
        ...
    #end
#end
```

- Add comments using the ## comment line marker to your custom templates to describe what changes you made compared to the default template the custom template was based on.
- Carefully read any velocity errors you might in your custom templates when generating your application.
- If you have problems getting your custom template to work correctly, you can temporarily add macro #MODEL_POINTER () to your template. It prints all “current” elements you can refer to. Here is an example of the output we get when we add this call at the top of our custom number slider template:

```
<!-- DEBUG:BEGIN:FORM_NUMBER_SLIDER : custom/item/form/formNumberSlider.vm, nesting level: 6 -->
<!--
=====
MODEL_POINTER
=====
Model pointer points to: PGItemModel (Salary)

JHS.current.group:          PGGroupModel (Employees)
JHS.current.tree:          -
JHS.current.item:          PGItemModel (Salary)
JHS.current.itemContainer:  PGGroupModel (Employees)
JHS.current.itemRegion:    -
JHS.current.regionContainer: -
JHS.current.include:       -
JHS.current.managedBean:   -
JHS.current.templateBindingsModel: PGItemModel (Salary)
JHS.current.menu:          -
JHS.current.groupFacesConfig: -
JHS.current.domainFacesConfig: -
JHS.current.pageComponent: PageModel (Employees)
JHS.current.domain:        -
=====
-->
<af:inputNumberSlider id="EmployeesSalary" value="#{bindings.EmployeesSalary.inputValue}"
...

```

12.4. Customizing Pages

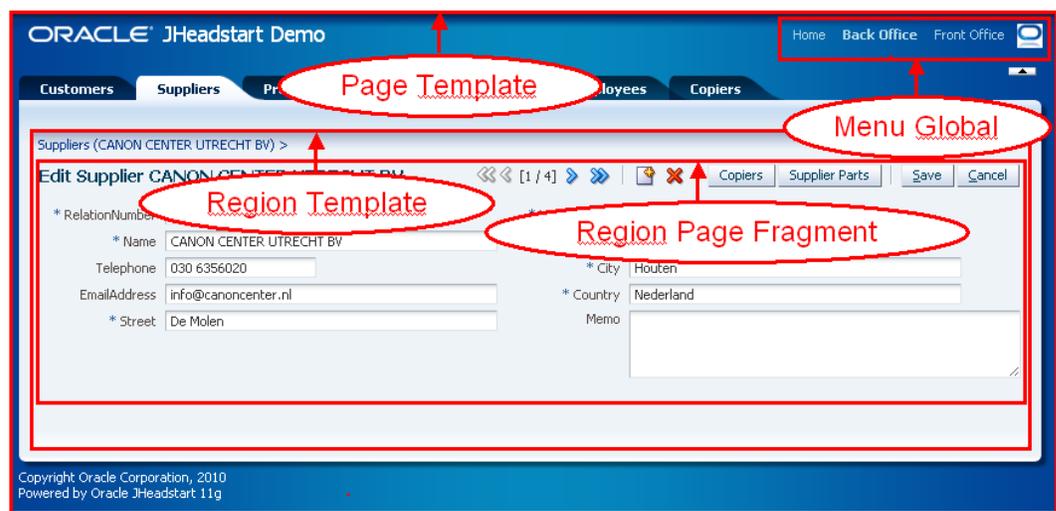
The previous general paragraph on creating custom templates already provides most of the information needed to customize generated pages. This paragraph discusses the following additional topics that are specific to customizing pages:

- Customizing Page Templates
- ADF Faces Skinning
- Changing the Page Template at Runtime
- Adding Custom Properties to a Generated Item
- Adding Custom Item Display Type

12.4.1. Customizing Page Templates

When generating groups as reusable regions with page fragments (recommended), an ADF Faces page has the following structure:

- The generated UIShell page uses a page template.
- The page template in turn uses a declarative component `menuGlobal.jsff` to provide the content of the upper right corner of the page.
- The page template has a `facetref` named `pageContent` that should be used to include the actual content of the page
- The generated UIShell page uses this `pageContent` facet to include a dynamic region that displays one of the generated group regions. The actual region displayed is determined through the menu.
- The region template, which only displays the breadcrumbs, and in case of a wizard layout the `af:train` component, has a `facetref` named `pageContent` that should be used to include the actual content of the page fragment.
- For each group, page fragments are generated that use the `pageContent` facet of the region template to include the actual page fragment content.



When using dynamic tabs (see chapter 9 "Generating menu Structures", section 9.3) the structure is slightly different. The generated dynamic tabs page template does not include a `pageContent` facetref. With dynamic tabs, the page template contains the declarative component `DynamicTabs.jsff` that can display up to 15 dynamic tabs, that are opened through the tree menu that is also contained by the page template.

The page template and the declarative components are generated using a velocity template, the actual content differs based on application-level settings like the type of menu used and security settings.

When generating the application, JHeadstart always generates three "sample" page templates:

- `JhsPageTemplate`: uses a tabbed menu at the top
- `JhsTreeMenuPageTemplate`: uses a tree menu at the left, and when clicking a menu item, the dynamic region at the right displays the group content for that menu item.
- `JhsDynamicTabsPageTemplate`: uses a tree menu at the left and dynamic tabs at the right.

Which of the "sample" page templates is actually used, is determined by the application level property **Page Template**.

Page Template *	/common/pageTemplates/JhsDynamicTabsPageTemplate.jspx
Region Template *	/common/pageTemplates/JhsPageTemplate.jspx
Common Pages Directory *	/common/pageTemplates/JhsTreeMenuPageTemplate.jspx
Menu Settings	{jhsLookAndFeelBean.currentPageTemplate}

So, to customize the page template, you actually have two options:

- Customize the generation of the sample page templates.
- Create your own page template.

12.4.1.1. Customizing Generation of the Sample Page Templates

You can change the generation of the sample page templates to suit your specific needs. You do this by creating custom templates for one or more of the templates used to generate the sample page templates and declarative components:

- `JHS_PAGE_TEMPLATE`
- `PAGE_TEMPLATE_CONTENT`
- `PAGE_TEMPLATE_TREE_MENU_CONTENT`
- `PAGE_TEMPLATE_DYNAMIC_TABS_CONTENT`
- `MENU_GLOBAL`
- `DYNAMIC_TABS`

Note that the `JHS_PAGE_TEMPLATE` is the "shell" Velocity template. Inside this template, one of the three page template content templates is parsed, based on the value of the **Page Template** property in the JHeadstart Application Definition Editor.

12.4.1.2. Create Your Own Page Template

You can make a copy of one of the generated sample page templates, rename it, and modify it as you like. Or you can create a page template from scratch. To use your custom non-generated template, you simply enter the path and name of your custom template in the **Page Template** property, instead of choosing one of the sample template names from the list.

Page Template *	/common/pageTemplates/myCustomPageTemplate.jsx
-----------------	--

There are some things to keep in mind when creating your own page template:

- When generating without dynamic tabs enabled, the generated shell pages assume that the page template contains a `pageContent` facetref, so make sure this facetref is still present in your custom template.
- Additional managed beans are generated when using the standard dynamic tabs page template. If you created your own custom template that still relies on these managed beans, you need to check the application-level property **Use Dynamic Tabs?**.



Attention: ADF registers page templates in an XML file named `pagetemplate-metadata.xml`, located in the `src/META-INF` directory. When you use the JDeveloper wizard to create a new page template, the template is automatically added to this XML file. However, if you want to base your page template on the default JHeadstart page template using “Save As...” you need to manually add your custom template to this XML file.

12.4.1.3. Customizing the Region Template

The region template is not generated. If you want to customize it, then make a copy, rename it and make your changes. In the **Region Template** property, specify the name of your custom region template.

Region Template *	/common/pageTemplates/myCustomRegionTemplate.jsx
-------------------	--

Make sure your custom region template contains a `pageContent` facetref and a `popups` facetref, since the generated page fragments use these facets.

12.4.2. ADF Faces Skinning

In addition to customizing page templates, you can customize the overall page look and feel by using a custom ADF Faces skin. 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, a Skinning Selectors document is available.



Web Reference: Web User Interface Developer’s Guide for Oracle ADF, chapter 20 “Customizing the appearance using styles and skins”.
http://download.oracle.com/docs/cd/E17904_01/web.1111/b31973/af_skin.htm#BAJFEFCJ



Web Reference: ADF Faces Skinning Selectors.
http://download.oracle.com/docs/cd/E15523_01/apirefs.1111/e15862/toc.htm

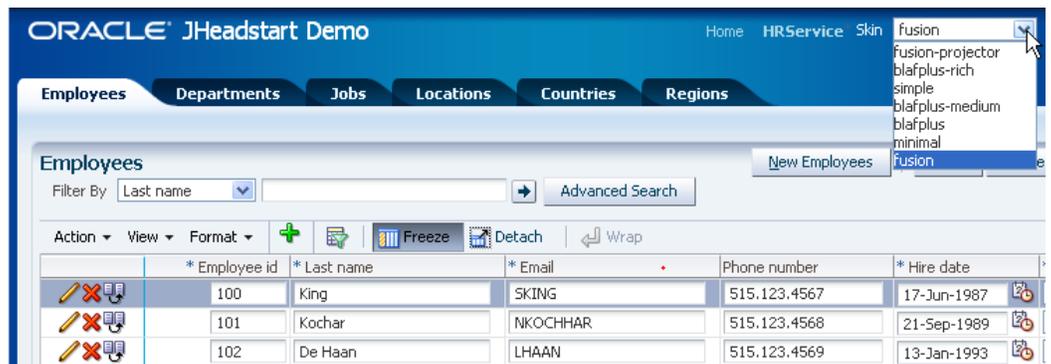
The skin used by ADF Faces is defined in a file named trinidad-config.xml, located in the WEB-INF directory.

```
<?xml version="1.0" encoding="windows-1252"?>
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
  <skin-family>fusion</skin-family>
</trinidad-config>
```

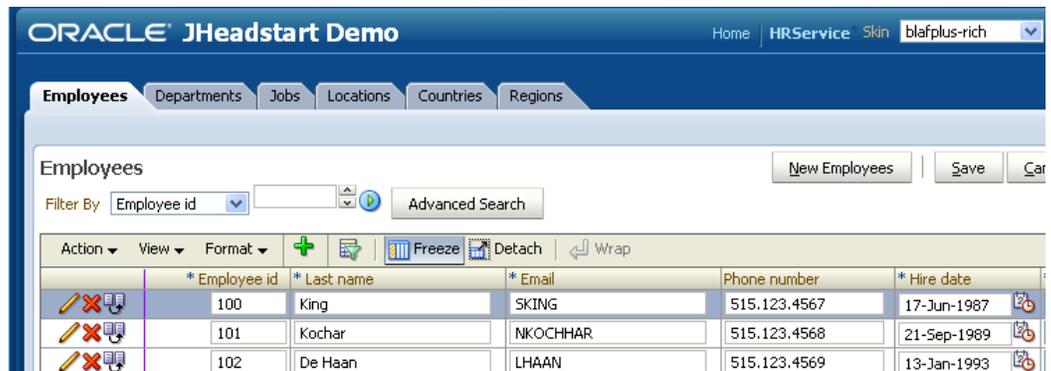
As you can see, the default skin is the “fusion” skin. The quickest way to see the differences between the various skins, is to generate your JHeadstart application with the application-level checkbox property **Generate UI Skin Switcher** checked.

Generate UI Skin Switcher? *

This generates a dropdown list in the upper-right corner of your page that by default will show all the predefined skins that are included with ADF Faces.



The most professional looking skins are the “fusion” skin (see above), and the “blafplus-rich” skin (see below).



If you define your own custom skin, it will automatically appear in the Skin dropdown list. When you generate a UI skin switcher, JHeadstart replaces the hardcoded skin-

family name in trinidad-config.xml with a reference to the “currentSkin” property of a managed bean named “jhsLookandFeelBean. The skin drop down list changes the value of the “currentSkin” property in this bean.

```
<?xml version="1.0" encoding="windows-1252"?>
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
  <skin-family>#{jhsLookandFeelBean.currentSkin}</skin-family>
</trinidad-config>
```

This “jhsLookandFeelBean” is defined in JhsCommon-beans.xml and references the LookAndFeelBean java class that is part of the JHeadstart Runtime library..

```
<managed-bean>
  <managed-bean-name>jhsLookandFeelBean</managed-bean-name>
  <managed-bean-class>oracle.jheadstart.controller.jsf.bean.LookAndFeelBean
</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>currentPageTemplate</property-name>
    <value>/common/pageTemplates/JhsPageTemplate.jspx</value>
  </managed-property>
  <managed-property>
    <property-name>currentRegionTemplate</property-name>
    <value>/common/pageTemplates/JhsRegionTemplate.jspx</value>
  </managed-property>
  <managed-property>
    <property-name>currentSkin</property-name>
    <value>fusion</value>
  </managed-property>
  <managed-property>
    <property-name>pageTemplates</property-name>
    <map-entries>
      <map-entry>
        <key>/common/pageTemplates/JhsPageTemplate.jspx</key>
        <value>Tabbed Menu</value>
      </map-entry>
      <map-entry>
        <key>/common/pageTemplates/JhsTreeMenuPageTemplate.jspx</key>
        <value>Tree Menu</value>
      </map-entry>
    </map-entries>
  </managed-property>
</managed-bean>
```

Through managed properties you can specify the (default) settings for the skin and page templates. To change these default values you can create a custom velocity generator template (not to confuse with ADF Faces page templates) for JhsCommonBeans.vm.

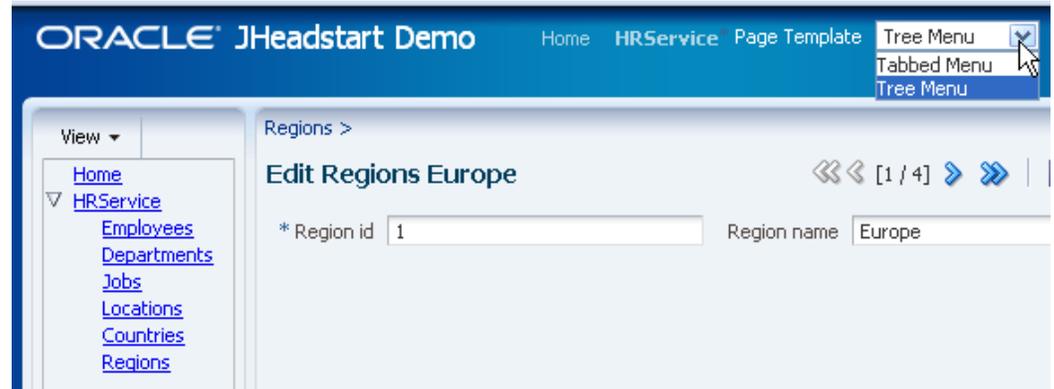
12.4.3. Changing the Page Template at Runtime

You can also change the page template at runtime, this might be useful for user interface prototyping, or when you want to use specific page templates based on the current ADF

faces skin. To make the page and region templates switchable at runtime, use the following property settings:

Page Template *	<code>#{jhsLookAndFeelBean.currentPageTemplate}</code>
Region Template *	<code>#{jhsLookAndFeelBean.currentRegionTemplate}</code>

By default, this will generate an additional drop-down list in the global menu area to switch the page template.



The page templates shown in this drop-down list are read from the “jhsLookAndFeel” bean already discussed in the previous section. To add your custom page templates to this list, you should make a custom template based on `JhsCommonBeans.vm` and change the “pageTemplates” property.

```
<managed-property>
  <property-name>pageTemplates</property-name>
  <map-entries>
    <map-entry>
      <key>/common/pageTemplates/JhsPageTemplate.jsp</key>
      <value>Tabbed Menu</value>
    </map-entry>
    <map-entry>
      <key>/common/pageTemplates/JhsTreeMenuPageTemplate.jsp</key>
      <value>Tree Menu</value>
    </map-entry>
    <map-entry>
      <key>/common/pageTemplates/MyCustomPageTemplate.jsp</key>
      <value>My template</value>
    </map-entry>
  </map-entries>
</managed-property>
```

If you don't want a separate page template drop-down list, but instead want to link the page template used to the selected skin, you can do the following:

- Leave the Page Template and Region Template to the above values that reference the `jhsLookAndFeel` bean.
- Make a custom Velocity template for `MENU_GLOBAL` (`menuGlobal.vm`) and remove the code that generates the Page Template dropdown list

- Make a custom Velocity template for JhsCommonBeans.vm and add an additional managed property named “skinPageTemplateMapping” to the managed bean definition. In this property you can specify which page template should be used with which skin. In the example below, the default tabbed menu is displayed with the fusion skin and the tree menu is displayed with the blafplus-rich skin.

```

<managed-property>
  <property-name>skinPageTemplateMapping</property-name>
  <map-entries>
    <map-entry>
      <key>fusion</key>
      <value>/common/pageTemplates/JhsPageTemplate.jsp</value>
    </map-entry>
    <map-entry>
      <key>blafplus-rich</key>
      <value>/common/pageTemplates/JhsTreeMenuPageTemplate.jsp</value>
    </map-entry>
  </map-entries>
</managed-property>

```

Note that you can take this concept even further by creating a subclass of the JHeadstart LookAndFeelbean. For example, you could add additional bean properties to specify the mapping between the skin and branding logo used.

12.4.4. Adding Custom Properties to a Generated Item

As explained in section 12.3.4.1 [Using Custom Properties](#), you can define custom properties against most element types in the JHeadstart Application Definition Editor and then use these properties in custom templates to write more reusable custom templates. For the item element, there is another alternative to make the template more generic and reusable, or even better, to stick with the default item template and still adding custom properties. This can be done using the item-level property **Additional Properties** which is only visible in expert-mode.

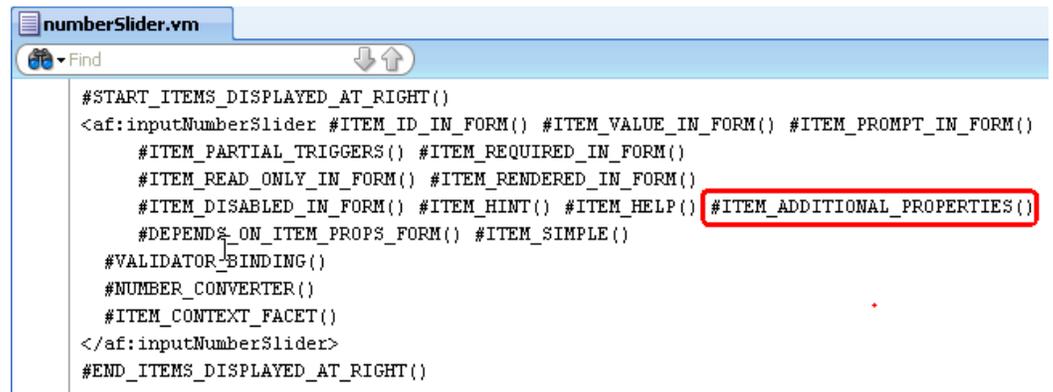
The value of this property is free text that will be added directly inside the ADF Faces Component generated for this item. You can use it to define additional properties like `styleClass` and `inlineStyle`. Use this property with caution, it might result in duplicate or invalid properties within an element. For example, when this item is a depends on item, the `autoSubmit` property will be generated, if you specify the `autoSubmit` property here as well, you will get XML parse exceptions at runtime.

All default item templates contain a call to `#ITEM_ADDITIONAL_PROPERTIES` macro which will add the content of this property when specified.

The screen shot below shows how we can use this property together with the generic custom `formNumberSlider.vm` template that we introduced in section 12.3.4 [Writing Reusable Custom Templates](#).

Customization Settings	
Additional Properties	maximum="25000" majorIncrement="1000" minimumIncrement="100" minorIncrement="100"

The `formNumberSlider.vm` template itself becomes simpler again, because it already contains a macro call to add the value of the **Additional Properties** property, as shown below.



```
numberSlider.vm
Find
#START_ITEMS_DISPLAYED_AT_RIGHT()
<af:inputNumberSlider #ITEM_ID_IN_FORM() #ITEM_VALUE_IN_FORM() #ITEM_PROMPT_IN_FORM()
    #ITEM_PARTIAL_TRIGGERS() #ITEM_REQUIRED_IN_FORM()
    #ITEM_READ_ONLY_IN_FORM() #ITEM_RENDERED_IN_FORM()
    #ITEM_DISABLED_IN_FORM() #ITEM_HINT() #ITEM_HELP() #ITEM_ADDITIONAL_PROPERTIES()
    #DEPENDS_ON_ITEM_PROPS_FORM() #ITEM_SIMPLE()
    #VALIDATOR_BINDING()
    #NUMBER_CONVERTER()
    #ITEM_CONTEXT_FACET()
</af:inputNumberSlider>
#END_ITEMS_DISPLAYED_AT_RIGHT()
```

12.4.5. Adding Custom Item Display Type

If you have created a custom item-level template, there is another way to apply this custom template: you can define a custom item display type that will show up in the item **Display Type** property list. You can associate the custom template with this custom display type, and JHeadstart will then pick up the custom template if you have specified the custom display type for an item.

We will use the slider number example from the previous paragraph to explain the steps to take, so we will introduce the custom display type `numberSlider`.

We first need to add this display type to the list of display types, which is configured in `jag-config.xml`. Open this file, and search for a bean named `viewTypeAdfFaces`. This bean has a property `supportedDisplayTypes` where we can add our custom display type `numberSlider`, as shown below.

```

<bean name="viewTypeAdfFaces"
      class="oracle.jheadstart.dt.share.model.ViewType"
      singleton="true">
  <property name="value">
    <value>adfFaces</value>
  </property>
  <property name="displayValue">
    <value>ADF Faces</value>
  </property>
  <property name="templateBindingFile">
    <value>defaultTemplateBindings.jtp</value>
  </property>
  <property name="supportedLayoutStyles">
    <list>
      <value>form</value>
      <value>table</value>
      <value>table-form</value>
      <value>select-form</value>
      <value>reusableTree</value>
      <value>tree</value>
      <value>tree-form</value>
      <value>parent-shuttle</value>
      <value>intersection-shuttle</value>
    </list>
  </property>
  <property name="supportedDisplayTypes">
    <list>
      <value>numberSlider</value>
      <value>textInput</value>
      <value>dropDownList</value>
      <value>lov</value>
      <value>model-choiceList</value>
    </list>
  </property>
</bean>

```

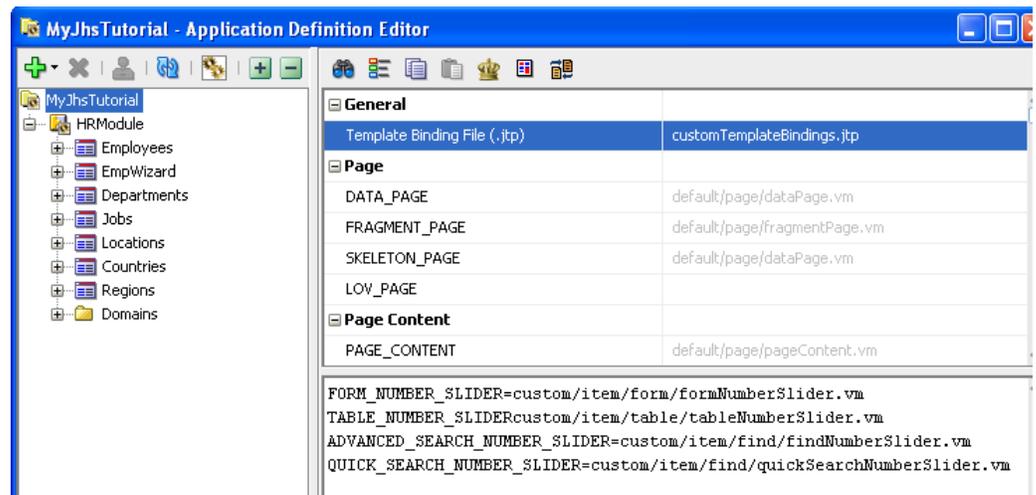
As explained before, JHeadstart creates an instance of `PGItemModel` for each item it generates. The properties injected during instantiation of a `PGItemModel` instance, including the templates to use, are also specified in `jag-config.xml`. JHeadstart looks up a bean named after the value of the display type, suffixed with `PGItemModel`. So, we need to create a new bean definition named `numberSliderPGItemModel` in `jag-config.xml`, that looks like this:

```

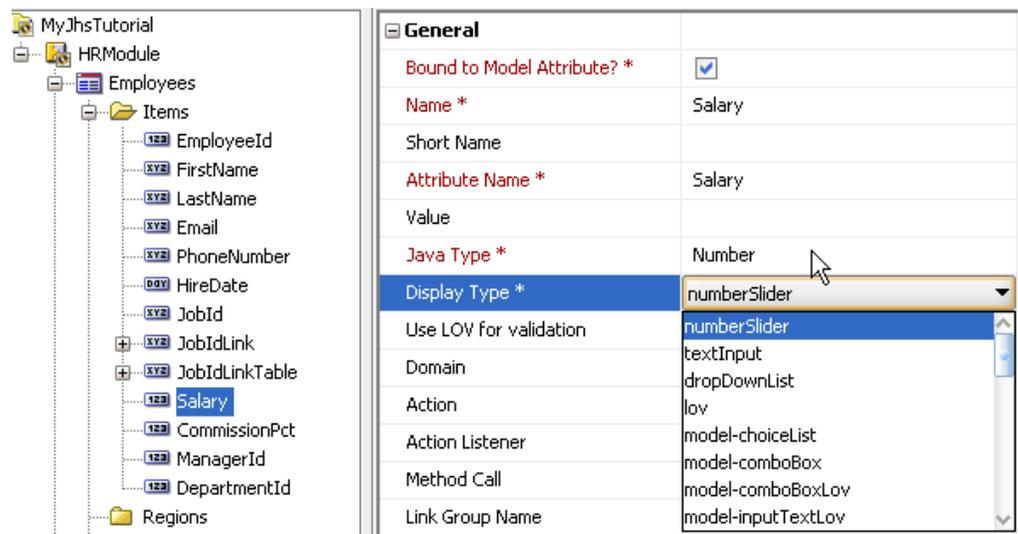
<bean name="numberSliderPGItemModel" class="oracle.jheadstart.dt.jag.pagegenerator.pgmodel.PGItemModel"
      singleton="false">
  <property name="formTemplateIdentifier"><value>FORM_NUMBER_SLIDER</value></property>
  <property name="tableTemplateIdentifier"><value>TABLE_NUMBER_SLIDER</value></property>
  <property name="advancedSearchTemplateIdentifier"><value>ADVANCED_SEARCH_NUMBER_SLIDER</value></property>
  <property name="quickSearchTemplateIdentifier"><value>QUICK_SEARCH_NUMBER_SLIDER</value></property>
  <property name="pgItemModelHelper"><ref bean="pgItemModelHelper"/></property>
</bean>

```

Now we need to add these logical template names in a custom template bindings file, and map them to physical templates. And we need to specify this custom template bindings file at the application level. See also section [Grouping Custom Templates](#).



Finally, we can set the custom display type for employee salary, and regenerate the application.



Warning: When you install a new JHeadstart version and re-enable JHeadstart on your project, the `jag-config.xml` file is overwritten again. So, after installing a new JHeadstart version you need to re-apply any changes you made to the `jag-config.xml` file.

12.5. Customizing Task Flows

This paragraph explains how to customize generated task flows. Note that as explained in section [Recommended Approach for Customizing JHeadstart Generator Output](#), we recommend that you first use the Visual ADF task flow diagrammer in JDeveloper to make any customizations. If your customizations work, then this paragraph explains how you can preserve these customizations when regenerating your application. This paragraph first explains the structure of the generated task flows which will make it easier to make your customizations.

12.5.1. Understanding Generated Task Flow Structure

JHeadstart generates one bounded task flow for each top-level group. The generated task flows are highly configurable to maximize reuse. They can be configured

- to start in create mode (new row)
 - to show one specific row (deeplinking)
 - to go to summary or detail page
 - to hide action buttons (Save, Cancel, Form brows Buttons)
 - to hide search region to show in read-only mode
- Standard task flow parameters**

To achieve this configurability, each task flow uses a task flow template. The task flow template defines a number of standard input parameters:

- `parentContext`
- `hideSaveButton`
- `hideCancelButton`
- `hideSearchArea`
- `hideFormBrowseButtons`
- `jhsBreadcrumbStack`

The `parentContext` parameter is automatically passed by JHeadstart. It contains a map object that you can use at your convenience. The map is created for the top-level unbounded task flow, and is passed to all child (region) task flows. It makes it very easy to share information between parent and child task flows, and even between sibling task flows, since they all have access to the same instance of the `parentContext` map. The `parentContext` parameter greatly reduces the number of use cases where you need to use the more complex mechanism of contextual events to implement region interaction. The drawback is that by relying on information in the `parentContext` map, the task flows become less reusable as they are tighter coupled to other task flows that put the required information in the `parentContext` map.

The `jhsBreadcrumbStack` is automatically passed by JHeadstart when you implement a `groupLink` where the new group task flow is shown "In Page". By passing the `jhsBreadcrumbStack`, JHeadstart ensures you will get breadcrumb links to navigate back to the calling group task flow.

The other template task flow parameters are Boolean parameters that can be used to conditionally show/hide buttons or the search area.

In addition, each generated task flow contains the following standard task flow parameters with a group name suffix:

- **create[Groupname]**: boolean parameter can be used to start the task flow in create mode: the form page will open with a new row ready for data entry. The other parameters can be used for deep linking
- **rowKeyValue[Groupname]**: value of primary key attribute, can only be used when there is one primary key attribute (which is recommended anyway). The value is used to call method `queryByKeyValue` on `JhsApplicationModule`.
- **rowKeyStr[Groupname]**: string representation of key, can be used for composite keys, and is used to call `setCurrentRowWithKey` on the iterator binding. If you want to pass the `rowKeyStr` value of the current row in a source page, you can use an EL expression like this:

```
{bindings.MyGroupNameIterator.currentRowKeyString}
```

- **rowKey[Groupname]**: the `oracle.jbo.Key` object that is used to call `findByKey` on the view object. If the primary key of your view object consists of multiple attributes, you must use this parameter. JHeadstart provides a convenience method to create this key. Suppose you need to deep link to an Employee which has a primary key consisting of both `FirstName` and `LastName`. The parameter value would become something like:

```
{jhsTypeConverter.listToJboKey['#{bindings.MyGroupEmpFirstName.inputValue},#{bindings.MyGroupEmpLastName.inputValue}']}
```

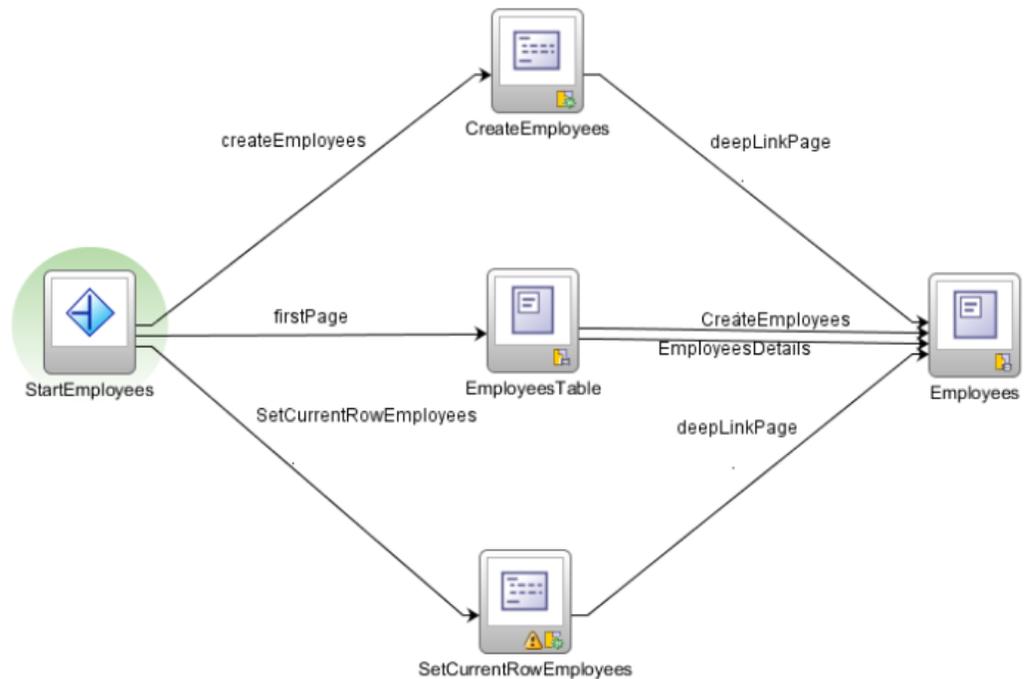
The function `listToJboKey` takes a comma delimited list as parameter, and converts that into an `oracle.jbo.Key`. Any parameter that is an EL-expression will be evaluated before it is put into the JBO key. If you want to pass the key of the current row in a source page, you can use an EL expression like this:

```
{bindings.MyGroupNameIterator.currentRow.key}
```

- **altKeyName[Groupname]**: the name of the alternative key, can be used in combination with `rowKey[Groupname]` to call `findByAltKey` on the view object.

12.5.1.2. Generated Task Flow Structure

The following picture provides a simplified view of the task flow generated for one top-level group with table-form layout.



The default activity of each generated task flow is always the "main router" activity that decides the next activity based on the values of the task flow parameters. By default the first page of the task flow will be shown. If the `create[Groupname]` parameter evaluates to true, a `CreateRow` activity will be executed followed by navigation to the "deeplink" page, which is the form page in case of a table-form layout. (In case of a table layout, the first page and deep link page are the same.)

If one of the row identifying parameters is set, the `SetCurrentRow` activity is invoked, followed by navigation to the deep link page.

Note however, that the above picture is a simplified view, when you look at the task flow diagram of a generated task flow, you will miss the direct lines between the various activities. This is because all the navigation rules have been generated as so-called wildcard control flow rules. This is required for implementing the breadcrumb navigation where the source view activity is unknown. It also makes it easier to change the default generated flow through the task flow, since you can navigate from any activity to any other activity. If you do not want this flexibility you can customize JHeadstart to generate more activity-specific control flow rules.

12.5.2. Customizing the Task Flow Template

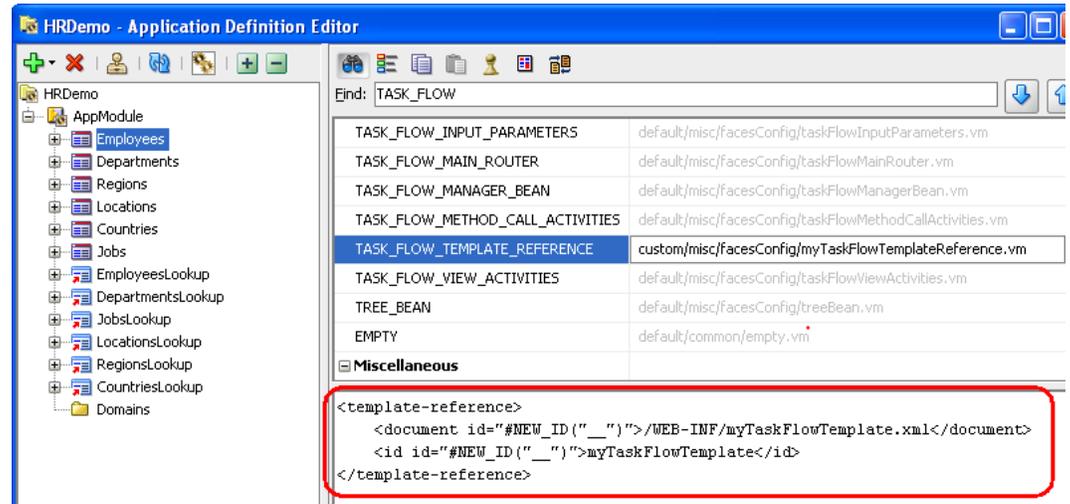
JHeadstart generates three task flow templates using the following template names

- `BOUNDED_TASKFLOW_TEMPLATE`: used to generate task flow template for task flows that contain stand-alone .jsf pages
- `FRAGMENT_TASKFLOW_TEMPLATE`: used to generate task flow template for task flows that contain .jsff page fragments
- `LOV_TASKFLOW_TEMPLATE`: used to generate task flow template for task flows generated from JHeadstart LOV groups

To customize the generation of these task flow templates, you can follow the standard procedure of defining a custom Velocity generator template as explained in more detail in section [Creating a Custom Template File](#).

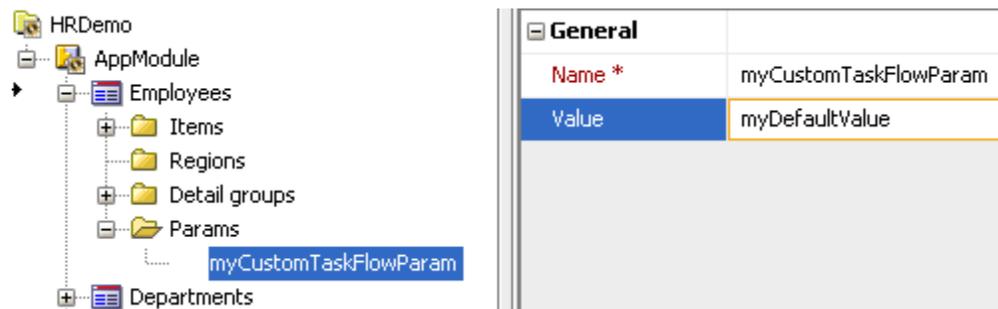
12.5.3. Customizing the Task Flow Template Reference

If you want base a generated task flow on a manually built task flow template rather than one of the generated task flow templates, you can create a custom Velocity template for the TASK_FLOW_TEMPLATE_REFERENCE template.



12.5.4. Adding Custom Task Flow Parameters

The task flow parameters section is generated through template TASK_FLOW_INPUT_PARAMETERS template. However, you do not need to create a custom template to add custom task flow parameters. You can add a custom task flow parameter to the generated task flow by defining a group parameter on a top-level group in the JHeadstart Application Definition editor. To add the parameter, right-click on the group and choose **New** -> **Parameter**. The value you specify for the parameter is used as the default value when invoking the task flow without specifying a value for the custom parameter. This default value can be a literal or an EL expression.



12.5.5. Adding Custom Managed Beans

To add custom managed beans to a task flow, you can create a custom Velocity template for the placeholder template TASK_FLOW_CUSTOM_BEANS, which by default is mapped to the default/common/empty.vm template.

Alternatively, you can implicitly add a custom managed bean by feeding the managed bean metadata model used by the task flow generator from another template. Typically, a managed bean is referenced from a page snippet, so in your custom page snippet template you can add the managed bean by calling `addCustomManagedBean` on the `JHeadstart FacesConfigGenerator`.

For example you can put this in the generator template where you want to add the managed bean generation (put it in one line, otherwise you will get an error):

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

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` method are as follows:

1. The Group used to determine in which task flow 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
6. Flag whether the custom bean should be generated within the bounded task flow (true) or outside of it (false)
7. Prefix for the scope of the bean, can be NULL.

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>
    pageFlow
  </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 `getProperty()`, you can include the following in the generator template where you called `addCustomManagedBean`:

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



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

12.5.6. Customizing the Main Router Activity

You can customize the generated main router activity by creating a custom Velocity template for the `TASK_FLOW_MAIN_ROUTER` template.

You can also implicitly customize the main router by feeding the `MainRouterCase` metadata model from another custom template. You do this by calling method `addMainRouterCase` on the `facesConfigGenerator`:

```
#{JHS.facesConfigGenerator.addMainRouterCase("#{someBooleanExpression}","SomeOutcome")}
```

The parameters of `addMainRouterCase` method are as follows:

1. The Group used to determine in which task flow the main router case should be added
2. The boolean expression used for the router case
3. The outcome for this router case

12.5.7. Adding Custom Task Flow Activities

To add custom activities to a task flow, you can create a custom Velocity template for the placeholder template `TASK_FLOW_CUSTOM_ACTIVITIES`, which by default is mapped to the `default/common/empty.vm` template.

When adding a custom activity, you also need additional control flow rules to navigate to and from the activity, and may be an additional main router case to conditionally go to the custom activity. You can create all these task flow artifacts within the custom activities template by feeding the main router case model, as well as the control flow rule model.

Here is an example:

```
<method-call id="SomeActivityId">
  <method id="#NEW_ID("__")">#{SomeMethodCall}</method>
  <outcome id="#NEW_ID("__")">
    <fixed-outcome id="#NEW_ID("__")">SomeOutcome</fixed-outcome>
  </outcome>
</method-call>
#{JHS.facesConfigGenerator.addMainRouterCase("#{someBooleanExpression}","SomeOutcome")}
#{JHS.facesConfigGenerator.addGlobalNavigationCase("#{JHS.current.group","SomeOutcome","SomeActivityId",false)}
#{JHS.facesConfigGenerator.addNavigationCase("#{JHS.current.group","SomeActivityId","SomeOutcome","SomeToActivityId",false)}
```

12.5.8. Adding Custom Control Flow Rules and Cases

To add custom control flow rules and cases to a task flow, you can create a custom Velocity template for the placeholder template `TASK_FLOW_CUSTOM_CONTROL_FLOW_RULES`, which by default is mapped to the `default/common/empty.vm` template.

Alternatively, you can implicitly add a control flow rules and cases by feeding the control flow rule metadata model used by the task flow generator from another template. You

will typically use this technique in the custom template where you add custom activities that need corresponding control flow rules.

To add the control flow rules from another generator template, you call `addNavigationCase` for a specific navigation between two activities, or `addGlobalNavigationCase` for a wildcard navigation on the JHeadstart `FacesConfigGenerator`.

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

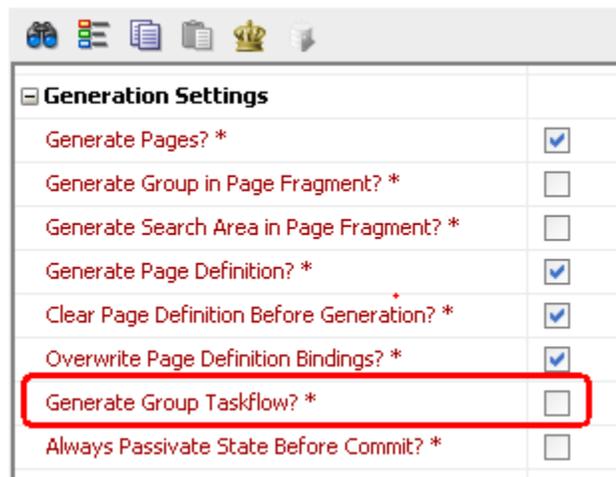
Similarly, `addGlobalNavigationCase` takes two parameters: just the outcome name and the "to" page.



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

12.5.9. Preventing Task Flow Generation

If you prefer to switch off generation of a specific task flow altogether, rather than using custom templates, you can do this using the group-level generator switch **Generate Group Taskflow?**.



Note that this property is only visible in expert mode.

12.6. Customizing Output of the File Generator

JHeadstart uses a special template, `default/misc/file/fileGenerator.vm`, as a means to generate additional files, not directly related to a group. Examples of these files are the home page, ADF Faces page and region templates, the login page, and SQL scripts to populate the JHeadstart database tables for table-driven features.

This section explains how to

- Customize the content of a file generated by the file generator
- Stop a file from being generated by the file generator
- Generate additional custom files.

12.6.1. How to customize the content of a file generated by the file generator

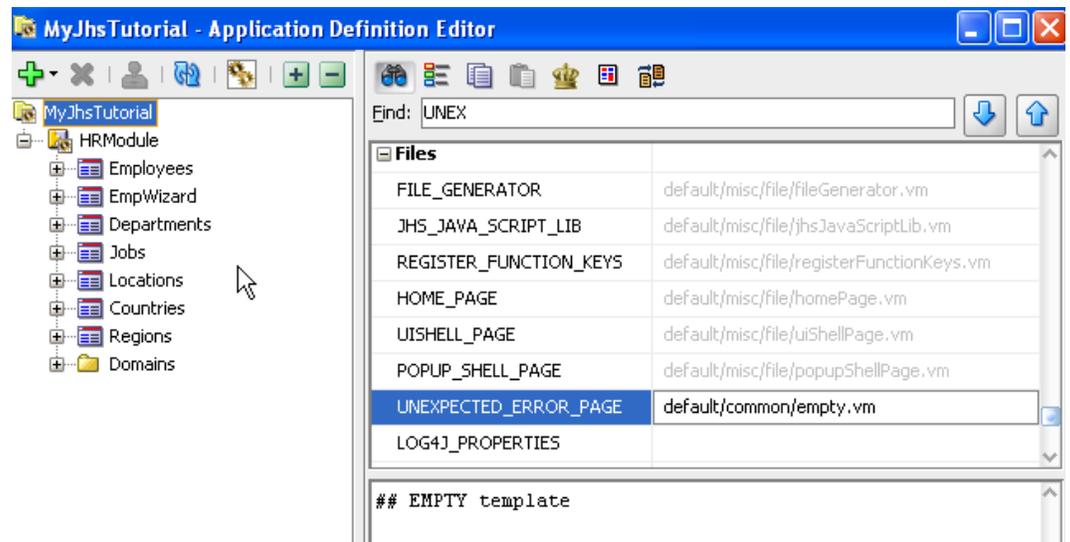
Each file that is generated through the `fileGenerator.vm` template has its own logical template name that is used to generate the file content. You can find these template names when opening up the `fileGenerator.vm` template.

```
## Generate Error Page
#set ($parsedContent = "#JHS_PARSE_NO_DEBUG( UNEXPECTED_ERROR_PAGE ${JHS.service})")
$JHS.createOrReplaceFile("${JHS.htmlRootDir}${JHS.application.commonPagesDir}UnexpectedError.jsff", $parsedContent)
```

Once you know the template name, you can create a custom template for it in the standard way, by going to the **Templates** tab of the JHeadstart Application Definition Editor and changing the name of the file template. See section [Creating a Custom Template File](#) for more info.

12.6.2. How to stop a file from being generated by the file generator

To stop a file from being generated, you specify the empty template shipped with JHeadstart as the "custom" template for this file template.



12.6.3. How to generate additional custom files

The `fileGenerator.vm` template contains a "place holder" template that you can use to generate additional custom files using the JHeadstart metadata model

```
## Generate custom files
#JHS_PARSE('CUSTOM_FILES' ${JHS.service})"
```

The `CUSTOM_FILES` template name by default maps to the `default/common/empty.vm` template. So, if you want to generate additional files you create your file generator template and specify this template as the custom template to use for `CUSTOM_FILES`.

You can use 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).

12.7. Customizing Page Definitions

12.7.1. Controlling Generation of Value Bindings

By default, JHeadstart will only generate value bindings needed for the items that are displayed. If the group generates a page with table layout, an `Item` binding inside the table tree `AttrNames` element will be generated.

```
<tree id="DepartmentsTable" IterBinding="DepartmentsIterator">
  <nodeDefinition Name="Departments" DefName="model.DepartmentsView">
    <AttrNames>
      <Item Value="DepartmentId" />
      <Item Value="DepartmentName" />
      <Item Value="ManagerId" />
      <Item Value="LocationId" />
    </AttrNames>
  </nodeDefinition>
</tree>
```

The value of such a binding can only be retrieved inside the `af:table` component using the following EL expression:

```
{row.bindings.[attrName].inputValue}
```

For example:

```
{row.bindings.DepartmentId.inputValue}
```

If the group generates a page with form layout an `attributeValues` binding will be generated.

```
<attributeValues id="DepartmentsDepartmentId"
  IterBinding="DepartmentsIterator">
  <AttrNames>
    <Item Value="DepartmentId" />
  </AttrNames>
</attributeValues>
```

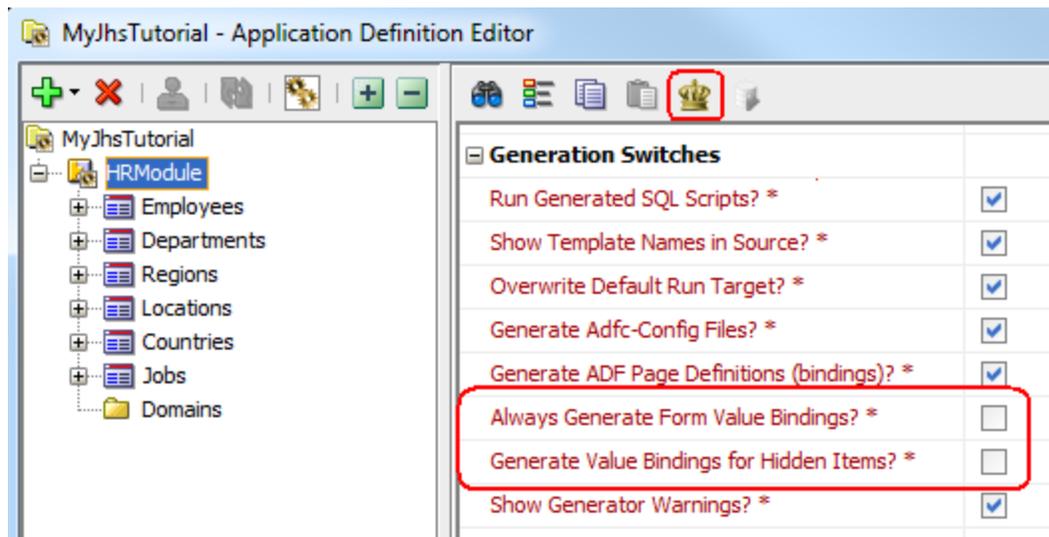
The value of such a binding can be using the following EL expression:

```
{bindings.[bindingId].inputValue}
```

For example:

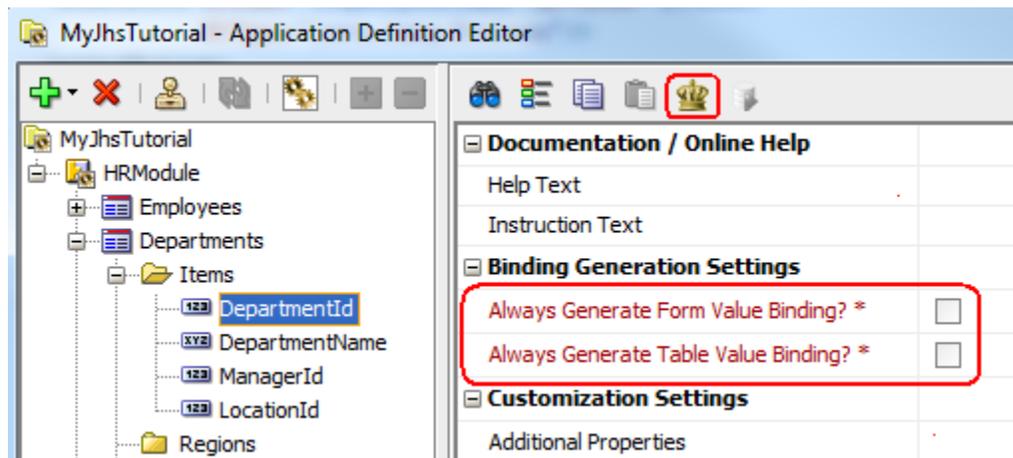
```
{bindings.DepartmentsDepartmentId.inputValue}
```

If you always want to generate form value bindings, even if a group does not generate a page with form layout (which was the standard behavior in JHeadstart versions prior to release 11.1.1.4.) you can check the service-level property **Always Generate Form Value Bindings**. Note that this property is only visible in expert mode.



If you want to generate a table and or form value binding for hidden items, (which again was the standard behavior in JHeadstart versions prior to release 11.1.1.4.) you can check the service-level property **Generate Value Bindings for Hidden Items**. Note that this property is only visible in expert mode.

If you want more fine-grained control over the generation of form value bindings, and generation of value bindings for hidden items, you should leave these two service-level properties unchecked, and use the item-level properties **Always Generate Form Value Binding** and **Always Generate Table Value Binding**.



12.7.2. Preserving Bindings Added using a Drag-and-drop Action

If you customize a generated page using drag and drop actions in the JDeveloper IDE, and you want to keep your application 100% generatable, you have two options to preserve the bindings that are added to the page definition during the drag and drop action. Firstly, you can uncheck the group-level generation switch **Clear page Definition Before Generation**. By unchecking this checkbox any content in the page definition that is not (re-)generated, will be preserved. The drawback of this approach is that if you have a large page definition with many bindings, then the re-generation of the page definition will become slower.

An alternative approach is to define the binding in a generator template and instruct the JHeadstart generator to add the binding to the page definition. Here is an example to generate a graph item including the required graph binding using a custom template:

```
#macro (CUSTOM_BINDING)
  <graph IterBinding="Employees4Iterator" id="EmployeesView4"
    xmlns="http://xmlns.oracle.com/adfm/dvt" type="BAR_VERT_CLUST">
    <graphDataMap leafOnly="true">
      <series>
        <data>
          <item value="Salary"/>
        </data>
      </series>
      <groups>
        <item value="LastName"/>
      </groups>
    </graphDataMap>
  </graph>
#end
${JHS.pageDefGenerator.addBinding(${JHS.page,"EmployeesView4","CUSTOM_BINDING()")}
<dvt:barGraph id="barGraph1"
  value="#(bindings.EmployeesView4.graphModel)"
  subType="BAR_VERT_CLUST"
  seriesRolloverBehavior="RB_HIGHLIGHT"
  animationOnDisplay="AUTO" threeDEffect="true">
  <dvt:background>
    <dvt:specialEffects/>
  </dvt:background>
  <dvt:graphPlotArea/>
  <dvt:seriesSet>
    <dvt:series/>
  </dvt:seriesSet>
  <dvt:o1Axis/>
  <dvt:y1Axis/>
  <dvt:legendArea automaticPlacement="AP_NEVER"/>
</dvt:barGraph>
```

As you can see, the XML content of the binding you want to add to the page definition is defined in a custom macro in the template. Then the binding is “registered” with JHeadstart by using the `addBinding` command on the `pageDefGenerator`. Note that the macro holding the binding content should be sequenced before the call to add the binding.

In a similar way, you can add parameters and executables to the page definition using the following commands:

```
${JHS.pageDefGenerator.addExecutable(${JHS.page,"MyIterator","CUSTOM_BINDING()")}
${JHS.pageDefGenerator.addParameter(${JHS.page,"MyParam","CUSTOM_BINDING()")}
```

When adding an executable, you can specify an additional argument after the custom binding macro reference which specifies the ID of another executable where the new executable should be sequenced before.

This page is intentionally left blank.

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.

ADF 11 offers many runtime customizations out-of-the-box.



See chapter 39 “Customizing Applications with MDS” and chapter 40 “Allowing User Customizations at Runtime” in the Fusion Developers Guide for ADF;

http://docs.oracle.com/cd/E24382_01/web.1112/e16182/customize.htm#CFHBABEB

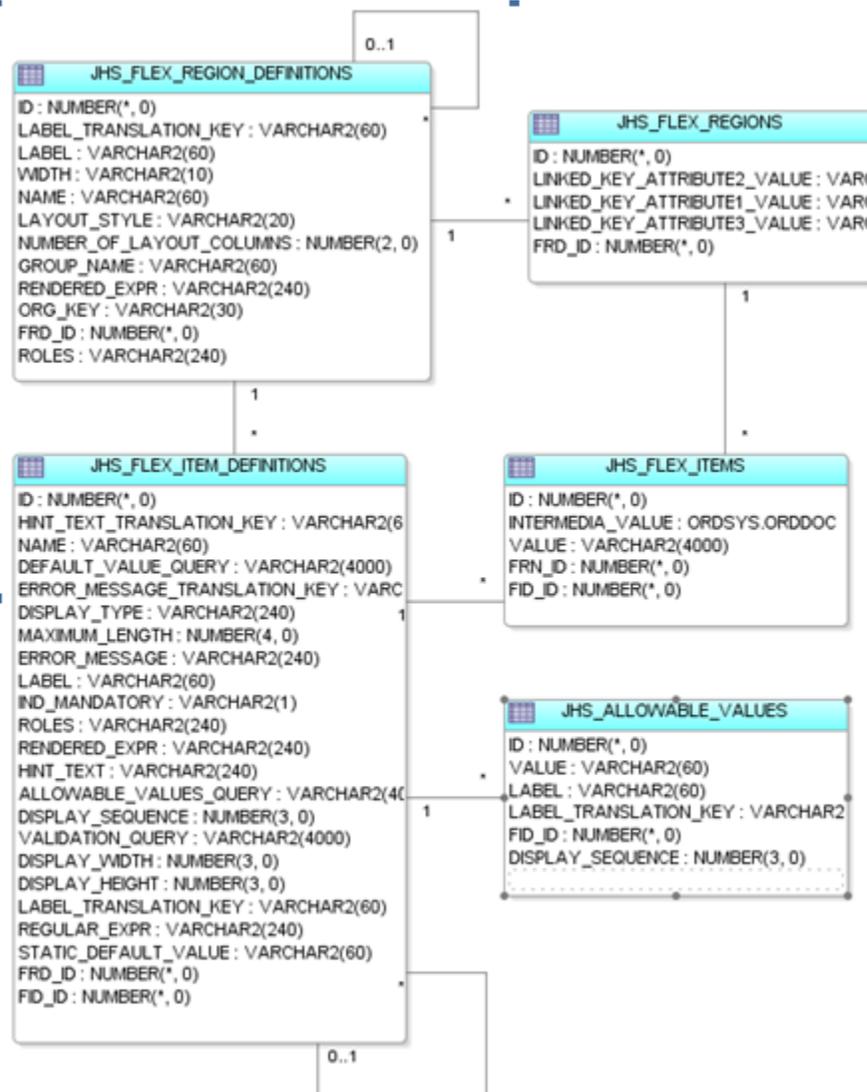
http://docs.oracle.com/cd/E24382_01/web.1112/e16182/ad_persist.htm#CIHHEHCF

If these features are not sufficient for you, you might want to leverage additional runtime page customization offered by JHeadstart through JHeadstart flex items. This feature is driven by a separate set of metadata stored in database-tables that come with JHeadstart, which allows you to support customer-specific requirements without changing the code base, and without changing the underlying database model. .

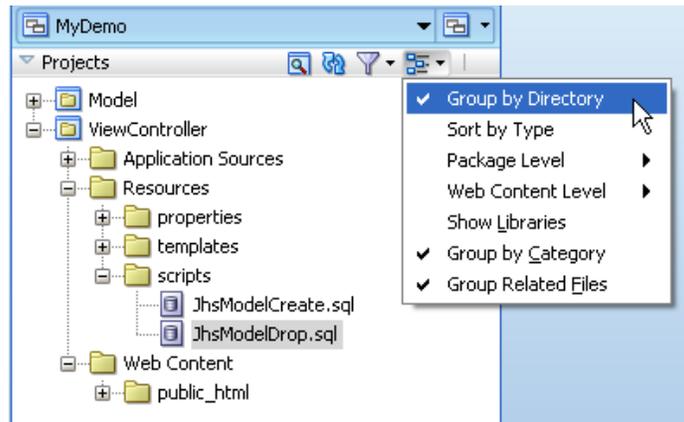
In the next sections we will explain how you can enable your application to use JHeadstart flex items.

13.1. 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 Group by Directories option 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.

13.2. Enabling Runtime Usage of Flex Items

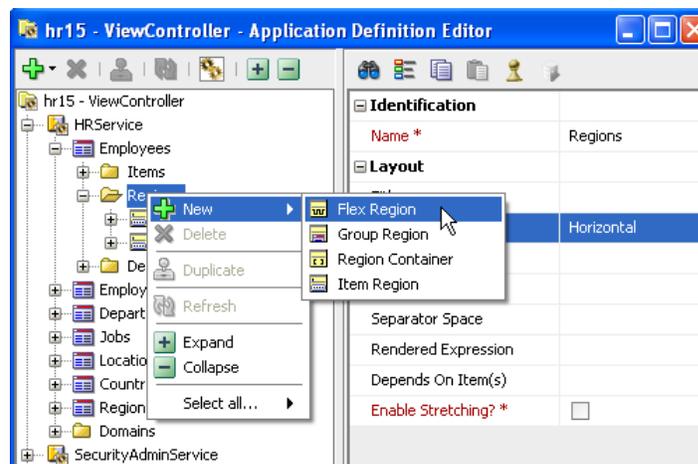
To enable your application for use of Flex Items, the first thing to do, is to check the *application*-level checkbox “**Allow Use of Flex Regions**”. Subsequently, in each JHeadstart *service* definition where you want to use flex items, you need to check the same checkbox property.



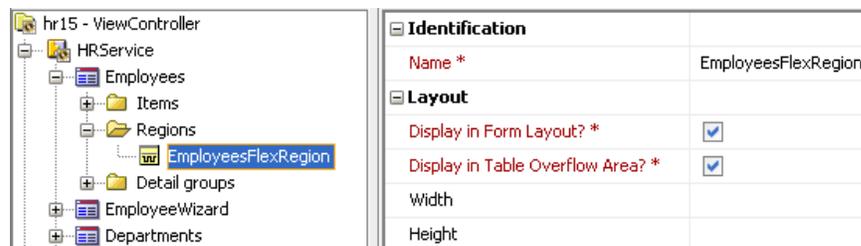
Next, you define “place holders” in the Application Definition Editor for a region of flex items that might be defined at runtime. A flex region placeholder can be defined in two ways: by creating a Flexible Region, or by creating an item with display type “flexRegion”.

13.2.1. 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.

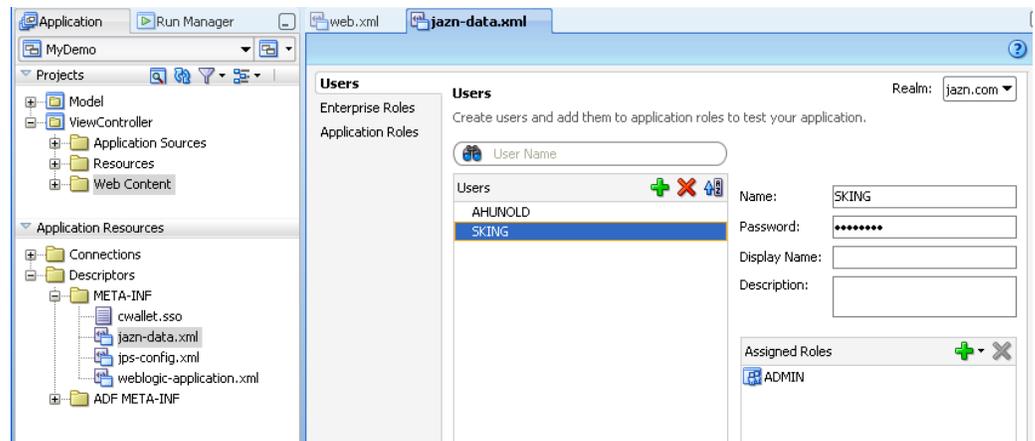


Note that you can define multiple Flex Regions for one group. For example, if you generate a wizard-style layout for your group, you might want to add a Flex Region to every wizard page.

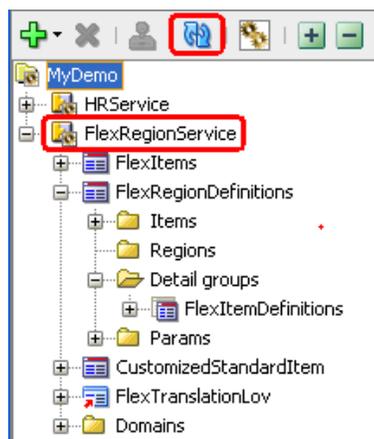
13.2.2. Running the JHeadstart Application Generator

When you now run the JHeadstart Application Generator again, the following happens to enable flex items at runtime:

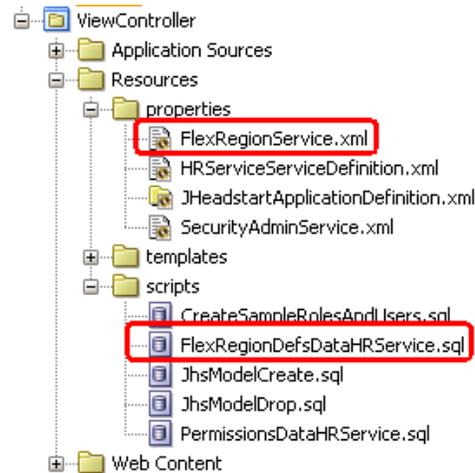
- All ADF Business Components included in the JHeadstart Runtime library are imported into your Model Project, and the **JhsModelService** application module, is added as a nested usage to your own application module. The **JhsModelService** includes View Object Usages that insert, update, delete and 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 FlexRegionService.xml is generated. Click Save All after running the JAG and then in the JHeadstart Application Definition editor, select the application level node, and click the synchronize button in the toolbar to see the new FlexRegionService.



- You can then click on the FlexRegionService in the Application Definition to generate the pages that are used to define the content of the Flex Region at runtime. This service 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 losing these changes when you regenerate your “own” service definition.
- A SQL Script named FlexRegionDefsDataServiceName.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.

13.3. 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.

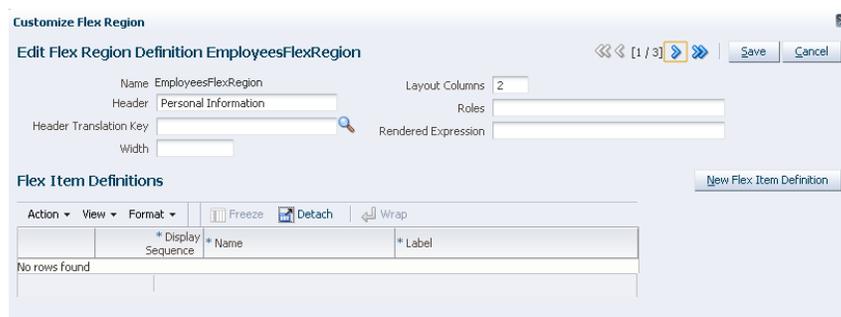


Note that if you have enabled role-based security in your application, you will only see the button when you log in as a user with a role as specified in the service-level property **Admin Role**. You can customize the appearance of the Customize Mode button, as well as the role required to see the button 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 FlexRegionService.



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 you 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 “tab” the flex region as shown below.

Edit Employees Kochhar

* EmployeeId: 101 * Email: NKOCHHAR
 FirstName: Neena PhoneNumber: 515.123.4568
 * LastName: Kochhar

Functional Information	Personal Information
Region of Birth: Europe	Favorite Color: <input type="radio"/> Red <input type="radio"/> Blue <input type="radio"/> Green
Country of Birth: Netherlands	Favorite Technologies: <input type="checkbox"/> ADF <input type="checkbox"/> JHeadstart <input type="checkbox"/> Struts
BirthDate: 30-Dec-1982	Curriculum Vitae: <input type="button" value="Choose File"/> No file chosen
Java Experience: <input type="text"/>	Private email: Neena.Kochhar@hotmail.com Neena.Kochhar@hotmail.com

Setting up another Item Region and setting the Layout Style of the Regions container to “tabbed” accomplish this.

The screenshot shows an IDE interface with a project tree on the left and a properties window on the right. In the project tree, the 'Regions' folder is expanded, showing two items: 'Functional' and 'FlexRegionEmployees', both of which are circled in red. The properties window for the 'Functional' item is open, showing the 'Layout Style' property set to 'Tabbed', which is also circled in red. Other properties like 'Name', 'Title', 'Width', and 'Height' are visible but not highlighted.

13.4. 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.

The screenshot shows the application definition tool interface. On the left, a tree view displays the application structure: MyDemo > HRService > Employees > Items > FlexRegionEmployees. The 'FlexRegionEmployees' item is selected. On the right, the configuration panel for this item is shown. The 'General' section includes: 'Bound to Model Attribute?' (unchecked), 'Name *' (FlexRegionEmployees), 'Short Name', 'Value', 'Java Type *' (String), and 'Display Type *' (flexRegion, highlighted with a red box). The 'Display Settings' section includes: 'Display in Form Layout? *' (true), 'Display in Table Layout? *' (false), 'Display in Table Overflow Area? *' (true), 'Display at Right of Item' (checked), 'Display Summary Type in Table', and 'Prompt in Form Layout'.

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.

The screenshot shows the 'Edit Employees Kochhar' application form. The form is divided into two columns. The left column contains standard form fields: EmployeeId (101), FirstName (Neena), LastName (Kochhar), Email (NKOCHHAR), PhoneNumber (515.123.4568), HireDate (21-Sep-1989), JobId (AD_VP), ManagerId (Davelaar), DepartmentId (Executive), and Salary (17000). The right column contains more complex fields: CommissionPct, Region of Birth (Europe), Country of Birth (Netherlands), BirthDate (30-Dec-1982), Java Experience, Favorite Color (radio buttons for Red, Blue, Green), Favorite Technologies (checkboxes for ADF, JHeadstart, Struts), Curriculum Vitae (Choose File button), and Private email (Neena.Kochhar@hotmail.com).

Note that you can define an unlimited number of **Flex Regions** and Items with **Display Type** “flexRegion” in a group.

13.5. 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 chapter 11 "Internalization and User Assistance" for more information on resource bundle types, and the option to change and translate your pages at runtime.

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.

This page is intentionally left blank.

Forms2ADF Generator

The JHeadstart Forms2ADF Generator (JFG) allows you to reuse Oracle Forms reuse Oracle Forms elements and properties when creating Oracle ADF applications. creating Oracle ADF applications.

The JFG creates the Business Services (ADF Business Components) and the JHeadstart meta data (Service 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.

14.1. 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.

The Forms2ADF generation process looks as follows:

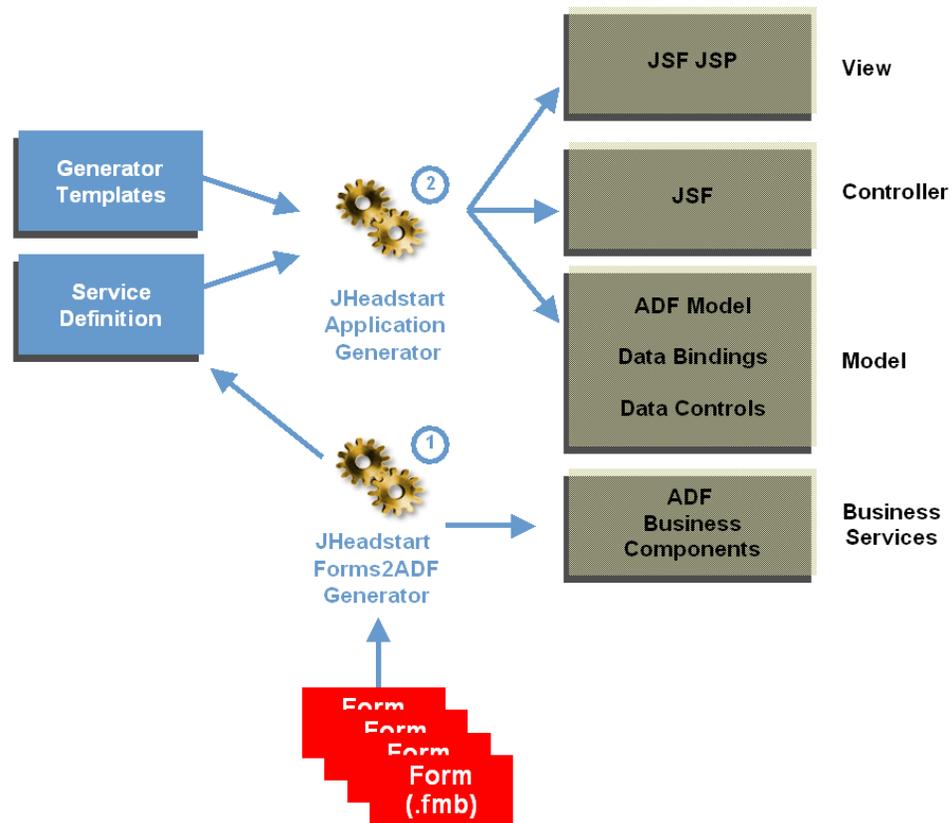


Figure 14-1 JHeadstart Forms2ADF Generator Process

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 View Accessors and List of Values elements are created for each LOV when Generator Setting “LOV Implementation” is set to “ADF Model”. See section 13.3.4 for more information on this setting.
 - ADF BC Application Modules are created for each form. All form-based application modules are nested inside one overall root application module.
2. JHeadstart Service 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 when Generator Setting “LOV Implementation” is set to “JHeadstart”. See section 13.3.4 for more information on this setting.
 - 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 more 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*”.

14.1.1. Added Value of JHeadstart Forms2ADF Generator

The amount of work that can be saved by using the JHeadstart Forms2ADF Generator very much depends on the structure of the Oracle Forms application at hand. The JHeadstart Forms2ADF Generator provides most savings for forms that have the following characteristics:

- Complex user interface, many (stacked) canvasses, many tabs, many list of values, and other display types
- Standard-Forms data retrieval and data manipulation through blocks based on database tables, with master-detail relations defined between the block
- PL/SQL logic mostly limited to user interface dynamics: conditionally showing/hiding user interface items, and conditionally changing the properties of user interface items. While JHeadstart does not convert PL/SQL logic, this type of logic is easily implemented in the ADF application because JHeadstart provides many declarative property settings to implement this behavior.

JHeadstart has made the deliberate choice to not automatically convert the PL/SQL logic to Java. The reasons for this are:

- It is impossible to automate the migration of a two-tier architecture (logic in Forms or in the database) to a three tier Model-View-Controller architecture as is common in JEE web applications, including ADF-based applications.
- The architecture of the converted application should be identical to the best-practice architecture of an ADF application that is build from scratch. If the architecture is the same, the same skill set can be used to maintain both migrated applications and ADF applications build from scratch. In addition, by going for a best practice architecture, the application is more flexible, and can be maintained easier at lower cost.
- When using the JHeadstart Forms2ADF Generator, you get this best-practice ADF architecture that is identical to ADF/JHeadstart applications that are built from scratch.

Note: Other Forms2ADF conversion tools currently available (December 2009) have taken a different approach. They focus on automating the conversion of PL/SQL to Java. The architecture of such a converted application is different from a best-practice JEE/ADF web application. The architecture is more Forms-like, sometimes even including typical forms constructs like "GO_BLOCK" in the converted Java source code. The user interface produced by these migration tools is often a Java applet. If your primary intent is to move away from Oracle Forms, and you are less concerned about the structure and architecture of the target Java application, then these other conversion tools might be a valid choice. However, if you want to migrate to a best-practice ADF architecture, then the JHeadstart Forms2ADF Generator is the best choice.

14.2. 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 the Forms2ADF Generator outputs**

Make sure you understand the meaning and content of the output produced by the JFG. See section 14.4 [Understanding the Outputs of the JHeadstart Forms2ADF Generator](#) for more information.

6. **Check, fix and enhance the migrated ADF Business Components.**

It is essential that the model project is valid and functionally correct before you start generating the web tier. See section 14.5.1 [Checking and Fixing the Model project](#) for more information.

7. **Check and enhance migrated JHeadstart service definition**

See section 14.5.2 [Checking and Enhancing the JHeadstart Service Definition](#) for more information.

8. **Inspect PL/SQL code in the form and determine whether and how to handle it.**

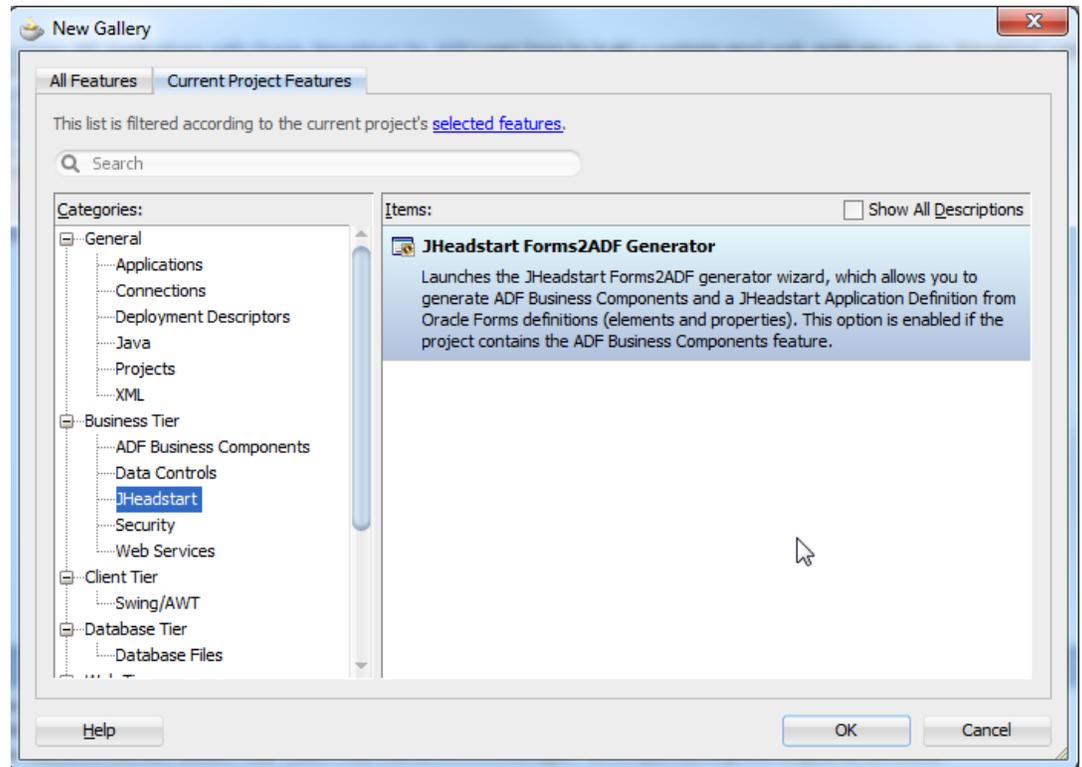
See section 14.5.3 [Handling Forms PL/SQL Logic](#) for more information.

9. **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.

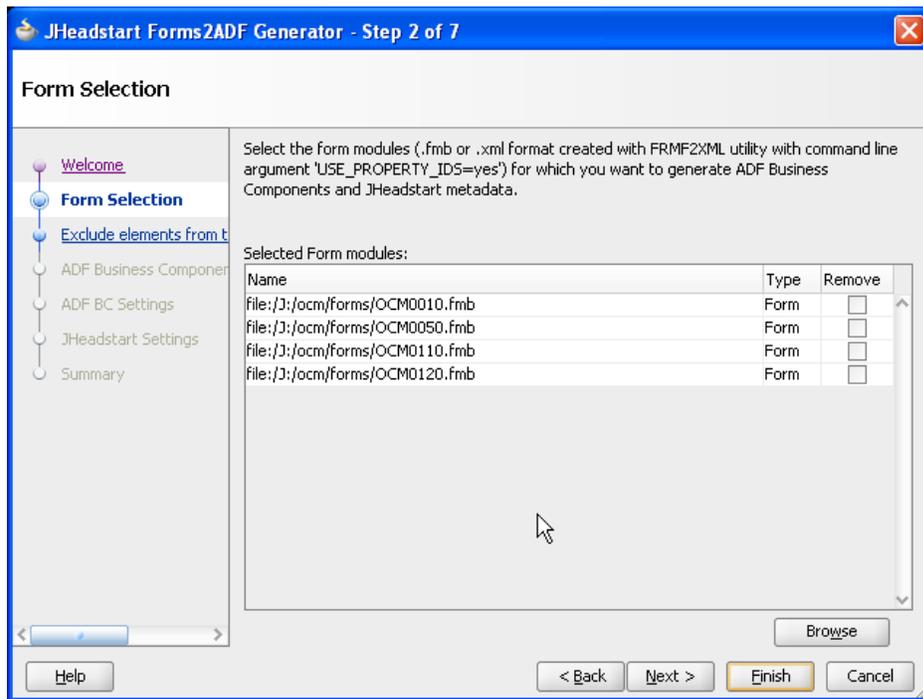
14.3. Running the JHeadstart Forms2ADF Generator (JFG)

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 JHeadstart node below the Business Tier. Select JHeadstart Forms2ADF Generator.



14.3.1. 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.



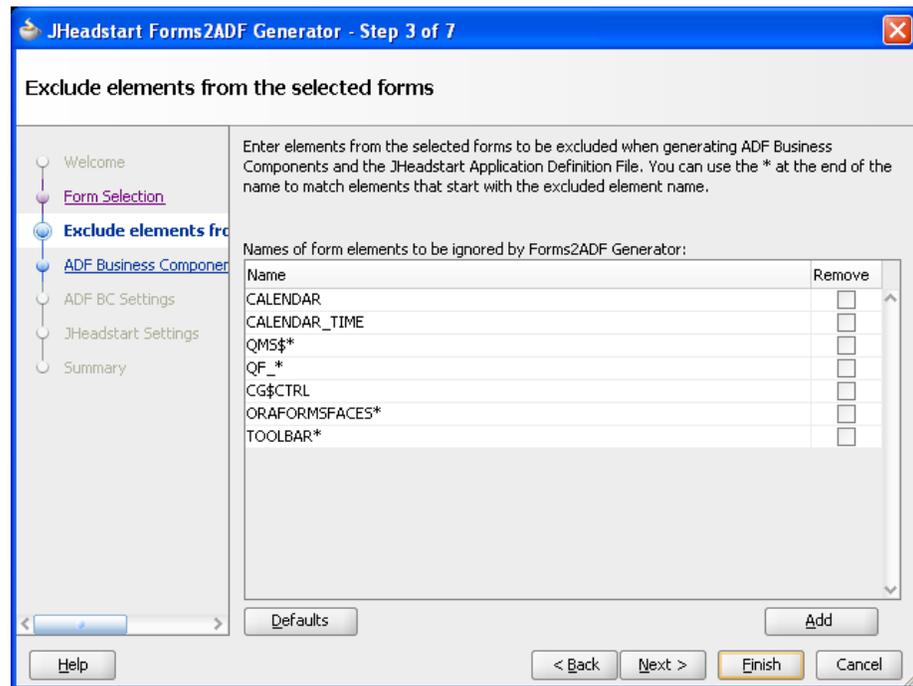
You can use the Browse button to open a File Chooser window to easily select the .fmb or .xml files from your file system.

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.

14.3.2. 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.
- TOOLBAR*: JHeadstart provides its own built-in functionality for toolbar functionality
-

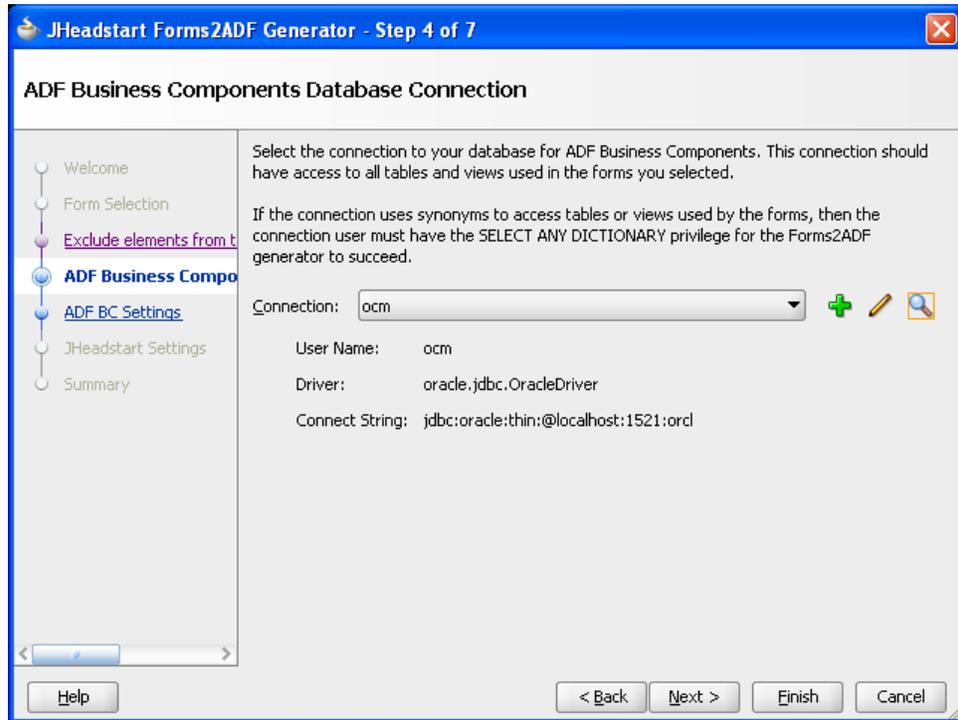


Reference: For information about OraFormsFaces, visit the website of Commit Consulting: <http://www.commit-consulting.com/oraformsfaces/>

14.3.3. Select Database Connection

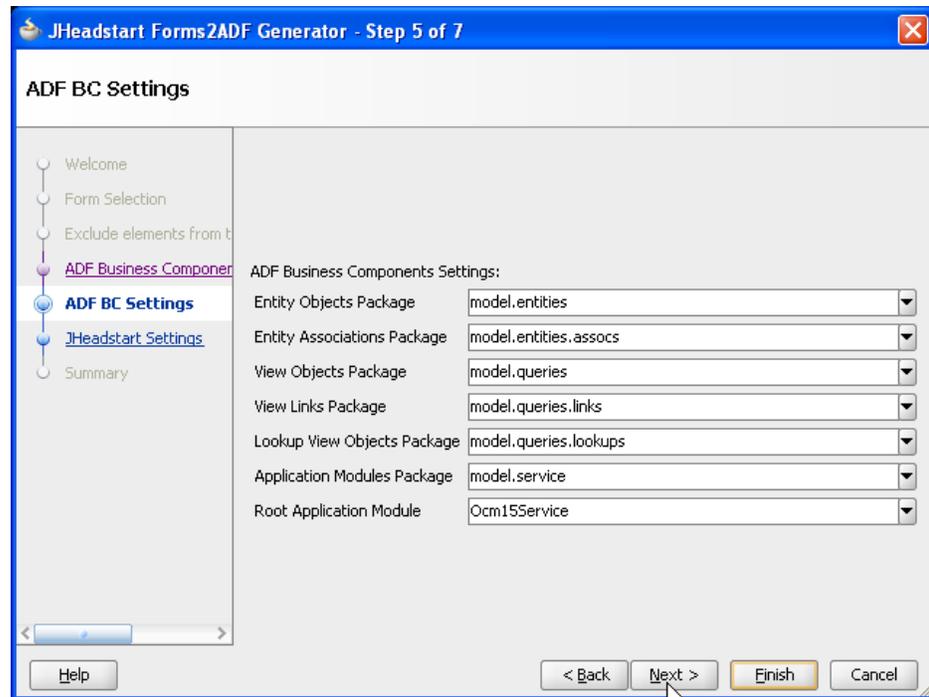
The database connection you select in this step is used as the connection for the ADF Business Components that will be generated. This connection is also used to query the

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 than 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.

14.3.4. ADF Business Components Settings

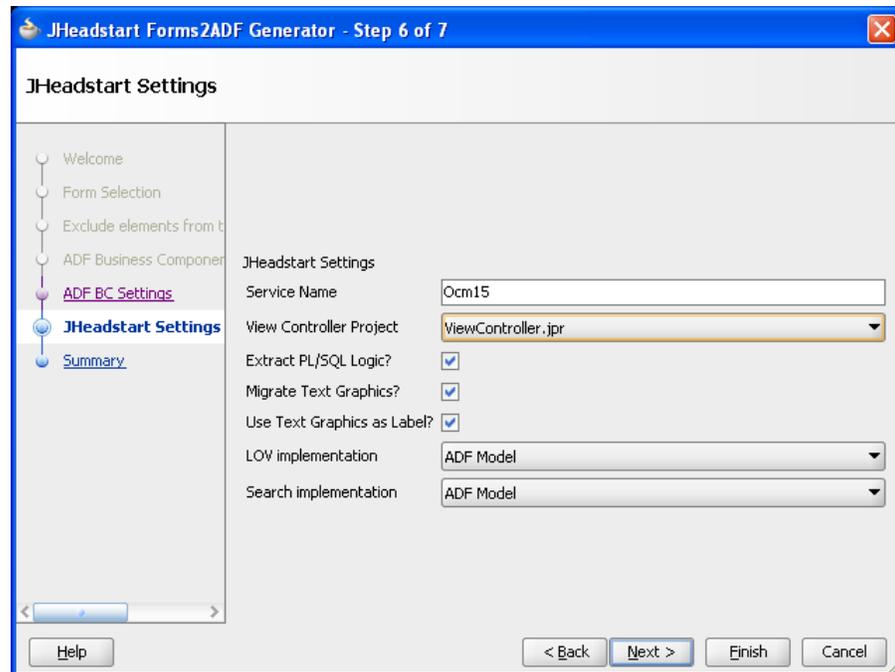


The following properties need to be set on the Generator Settings panel:

- **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.
- **Entity Associations Package:** The package name that will be used to store all entity associations. 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 created based on Oracle Forms blocks.
- **View Links Package:** The name of the package that will be used to store all view links, created based on relations between Oracle Forms blocks.
- **LookupView Objects Package:** The name of the package that will be used to store all view objects created based on Oracle Forms record group queries.
- **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.
- **LOV Implementation:** When choosing “ADF Model”, the JFG will create model-based List of Values within the ADF Business Components. When choosing “JHeadstart”, the JFG will create LOV groups and LOV item usages in the JHeadstart Service Definition file.
- **Search Implementation:** When choosing “ADF Model”, the JFG will set the Group-level quick and advanced search properties in the Service Definition to “model”. When choosing “JHeadstart”, these properties will be set to “samePage” for advanced search and “dropDownList” for quick search. It is recommended to use the same values for **LOV Implementation** and **Search Implementation**. This is because in an ADF Model-based search area, you can only include ADF Model based LOV items, and vice versa: in a JHeadstart quick or advanced search area, you can only include JHeadstart Lov items.

14.3.5. JHeadstart Settings



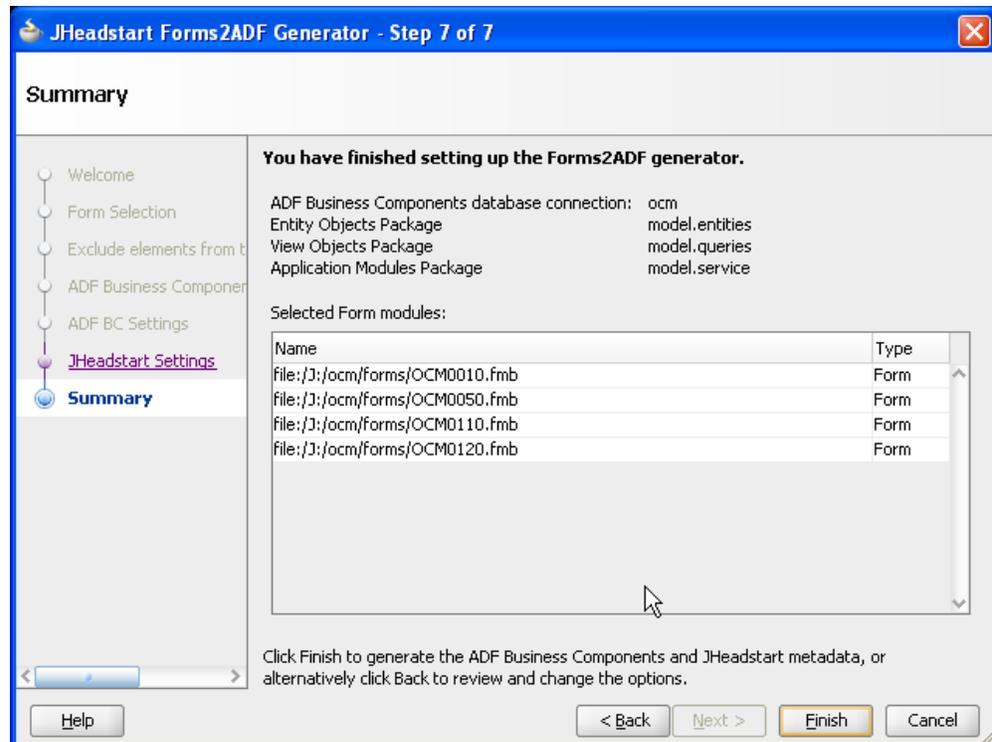
The following properties need to be set on the Generator Settings panel:

- **Service Name:** This is the name that is used to create the JHeadstart Application Definition file
- **View Controller Project:** The name of the project in which the JHeadstart Application Definition file will be saved.
- **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.
- **Migrate Text Graphics?:** If this checkbox is checked, any boilerplate text created through text graphics will be migrated. A text graphic migrates to an unbound read-only item with the value set to the text of the item.
- **Use Text Graphics as Label?:** If you check this checkbox and the **Migrate Text Graphics?** checkbox is checked, then a text graphic that is displayed at the left of an input item, without any other item in between, will be used as the label of the input item. In other words, such text graphics are not migrated in itself, instead the text is set as label of the item positioned at the right of the text graphic. Use this setting when converting older versions of Oracle Forms that did not yet support the label property on input items.
- **LOV Implementation:** When choosing "ADF Model", the JFG will create model-based List of Values within the ADF Business Components. When choosing "JHeadstart", the JFG will create LOV groups and LOV item usages in the JHeadstart Service Definition file.

- **Search Implementation:** When choosing “ADF Model”, the JFG will set the Group-level quick and advanced search properties in the Service Definition to “model”. When choosing “JHeadstart”, these properties will be set to “samePage” for advanced search and “dropDownList” for quick search. It is recommended to use the same values for **LOV Implementation** and **Search Implementation**. This is because in an ADF Model-based search area, you can only include ADF Model based LOV items, and vice versa: in a JHeadstart quick or advanced search area, you can only include JHeadstart Lov items.

14.3.6. Processing the Selected Forms

When you click the Finish button on the Summary panel, the JFG will start processing the forms.

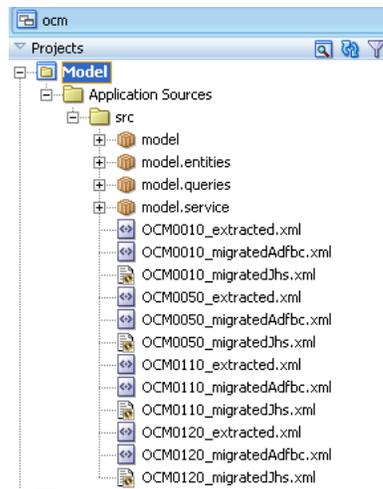


The processing consists of the following steps:

- **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 JFG run was successful, you can safely delete the intermediate results of the JFG, 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.



14.3.7. 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.

```

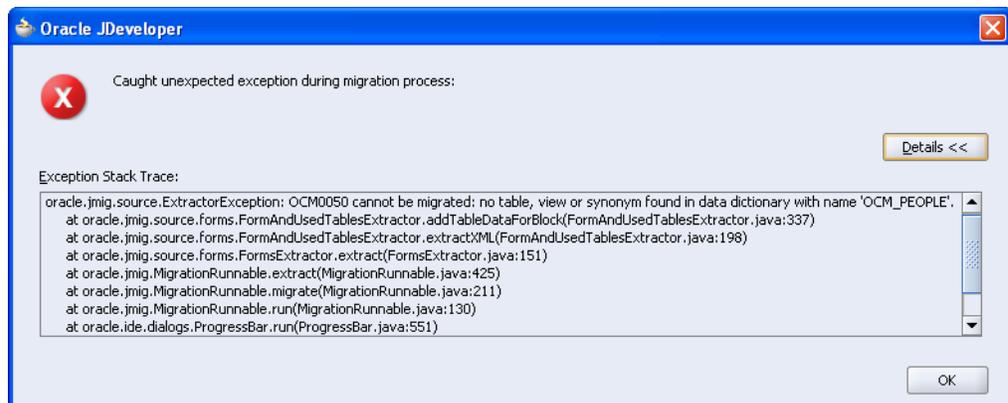
JHeadstart Forms2ADF Generator - Log
INFORMATION: Extracting file:/J:/ocm/forms/OCM0010.fmb
INFORMATION: Extracting file:/J:/ocm/forms/OCM0050.fmb
INFORMATION: Extracting file:/J:/ocm/forms/OCM0110.fmb
INFORMATION: Extracting file:/J:/ocm/forms/OCM0120.fmb
INFORMATION: Creating ADF BC migration XML for OCM0010
INFORMATION: Creating ADF BC migration XML for OCM0050
INFORMATION: Creating ADF BC migration XML for OCM0110
INFORMATION: Creating ADF BC migration XML for OCM0120
INFORMATION: Composing ADF Business Components for OCM0010
INFORMATION: Composing ADF Business Components for OCM0050
INFORMATION: Composing ADF Business Components for OCM0110
INFORMATION: Composing ADF Business Components for OCM0120
INFORMATION: Creating JHeadstart migration XML for OCM0010
INFORMATION: Creating JHeadstart migration XML for OCM0050
INFORMATION: Creating JHeadstart migration XML for OCM0110
INFORMATION: Creating JHeadstart migration XML for OCM0120
INFORMATION: Composing JHeadstart Application Definition for OCM0010
INFORMATION: Composing JHeadstart Application Definition for OCM0050
INFORMATION: Composing JHeadstart Application Definition for OCM0110
INFORMATION: Composing JHeadstart Application Definition for OCM0120

```

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 14.3.3 “Select Database Connection” for more information.

While you can fix the above error, there might be other unexpected errors that are not easily solved. 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.

14.3.8. 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.

14.4. Understanding the Outputs of the JHeadstart Forms2ADF Generator

This section will give you an overview of the output and explains how this can be mapped to Oracle Forms elements and their properties. It also discusses possible changes you might need to make to the generated outputs.

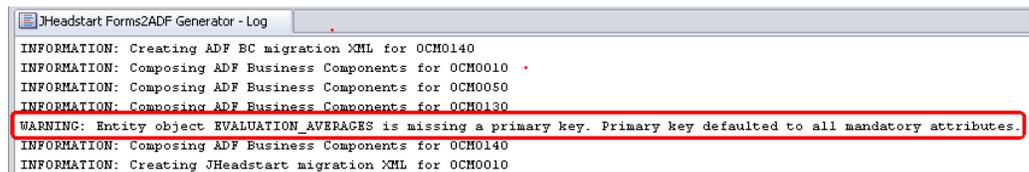
14.4.1. Generated ADF Business Components

14.4.1.1. 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:



```
JHeadstart Forms2ADF Generator - Log
INFORMATION: Creating ADF BC migration XML for OCH0140
INFORMATION: Composing ADF Business Components for OCH0010
INFORMATION: Composing ADF Business Components for OCH0050
INFORMATION: Composing ADF Business Components for OCH0130
WARNING: Entity object EVALUATION_AVERAGES is missing a primary key. Primary key defaulted to all mandatory attributes.
INFORMATION: Composing ADF Business Components for OCH0140
INFORMATION: Creating JHeadstart migration XML for OCH0010
```

When you 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.

14.4.1.2. 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.

14.4.1.3. View Objects

View Objects (VO) are created for each block in the form with the property **Query Data Source Type** set to "Table" or "FROM clause query". The values "Transactional Triggers" and "Procedure" are currently not supported.

When set to "Table", the VO is based on the entity object that maps to the table or view as specified in the **Query Data Source Name** property.

When set to "FROM Clause Query", the view object is created as a read-only view object, not based on an entity object. Note that this might require manual correction afterwards:

if the **DML Data Source Type** is set, the block is updateable, and the View Object should be based on the entity object that maps to the table as specified in the **DML Data Source Name** property. In this case, you should manually re-map the existing read-only attributes to the corresponding underlying entity object attribute. You can do this on the Attribute Mappings pane of the Edit Query dialog.

In addition to the VO's created for data blocks, read-only VO's are created for each Record Group Query found in the form that is tied to a List of Values object used by a migrated item.

The WHERE and ORDER BY clauses of the data blocks are also migrated to the View Object.

Any references in the SQL Query, the WHERE clause or ORDER BY clause 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.

14.4.1.4. View Object Attributes

A View Object attribute is created for each data bound item (an item based on a column) within the block, and for each unbound item that is populated through an LOV.

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 on an unbound item 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.

The table below shows the mapping between View Object Attribute properties and related Item properties in the form.

Attribute Property	Derived from Item Property	Comments
Name	Item Name	Item name is "CamelCapped", reserved words are suffixed with "1"
Updateable	Insert Allowed, Update Allowed	Set to true when both properties are true, set to false when both item properties are false, set to "While New" when only Insert Allowed is true. If the item is not visible, it is always made updateable.

Mandatory	Required	Value copied when attribute is not based on entity attribute
Value	Initial Value	Value copied as literal, except when Default Value is (variation of) <code>\$\$DATE\$\$</code> , then the Value Type is set to "Expression" and the Value to <code>"adf.currentDate"</code> or <code>"adf.currentDateTime"</code> .
Queryable	Query Allowed	
Precision	Maximum Length	Only set when attribute not based on entity attribute
Type	Data Type	Only set when attribute not based on entity attribute, SQL type converted to Java Type
Control Hints		
Display Hint	Visible	
Label Text	Prompt	
Tooltip Text	Tooltip, Hint	Copied from Tooltip property when set, otherwise copied from Hint property
Display Width	Width	Conversion algorithm to Display Width value based on Forms Coordinate System.
Display Height	Height	Conversion algorithm to Display Height value based on Forms Coordinate System.

14.4.1.5. View Accessor and List of Values

If you have chosen "ADF Model" as LOV Implementation, then for each LOV that is based on a Record group Query, and linked to a databound item, a View Accessor element will be created in the View Object, and a List of Values definition will be created in the View Object attribute that was created for the LOV item. For each LOVColumnMapping in the Form LOV a List Return Value is created in the ADF Model List of Values.

14.4.1.6. 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 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 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, the JFG will display a warning in the log window:

```
JHeadstart Forms2ADF Generator (Preview) - Log
INFORMATION: Creating ADF BC migration XML for ZRG3024F
INFORMATION: Creating ADF BC migration XML for ZRG5009F
WARNING: Cannot find element with ID 'PGE_VERANTWOORDINGEN_VW.PCM_AWP_CODE2'
WARNING: Cannot find matching destination attribute for join condition part 'VTC2.PCM_AWP_CODE2' in block 'AVE',
view link destination attribute(s) for view link Zrg5009fCgfkSpgbVawFkAve not created or incomplete.
WARNING: Cannot find element with ID 'PGE_VERANTWOORDINGEN_VW.PCM_AWP_REL_NR2'
WARNING: Cannot find matching destination attribute for join condition part 'VTC2.PCM_AWP_REL_NR2' in block 'AVE',
view link destination attribute(s) for view link Zrg5009fCgfkSpgbVawFkAve not created or incomplete.
INFORMATION: Creating ADF BC migration XML for ZRG6003Q
INFORMATION: Composing ADF Business Components for REL1023F
```

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.

14.4.1.7. Application Module

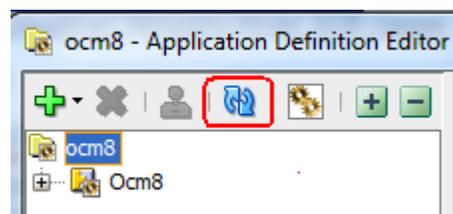
For each form, an application module is created that contains usages for all the View Objects created for the form. Each form-specific application module is added as a nested application module usage to the Root Application Module.

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 whether:

- your project compiles successfully. If not correct the compilation errors first.
- all View Object queries have been brought forward in a correct manner, and return the correct data
- you can make updates through the entity-based View Objects, which are all View Objects that originate from a Forms block.

14.4.2. Generated JHeadstart Service Definition File

The JFG also creates the JHeadstart Service Definition file. If you open the JHeadstart Application Definition editor and the new service is not yet visible, the select the application node, and click the synchronize button.



The sub sections below explain how and when each element type in the JHeadstart Application Definition file is created.

14.4.2.1. 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 property **Group Usage** set to “List of Values Window”.

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.

14.4.2.2. 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.

14.4.2.3. Item List of Values

If you have chosen “JHeadstart” as LOV Implementation, then for each LOV that is based on a Record group Query, and linked to a databound item, a List of Values item will be created as a child of the JHeadstart item that was created for the LOV item in the form.

For each LOVColumnMapping in the Form LOV a List of Values Return Value is created in the JHeadstart Application Definition file.

14.4.2.4. 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.

14.4.2.5. 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.

14.5. Common Steps After Running the Forms2ADF Generator

This paragraph provides a checklist of common steps you should consider to perform after you migrated a form using the Forms2ADF generator.

14.5.1. Checking and Fixing the Model project

Before generating the web application using the JHeadstart Application Generator, you should ensure you have a valid and correctly working set of ADF Business Component in the Model project.

- Fix any incomplete view links as reported in the log window.
- Check all view objects for use of bind parameters that need to get a value. Determine how you will supply these bind variable values.
- Check application data model: are master-detail relations properly carried forward in the data model? If not, then add the required view links and change the data model to get the nested view object usages that reflect the view object usages.
- Compile the Model project: are there any errors or warnings that need attention
- Run the application module tester: do all queries work fine? Do all model-based list of values work correctly, and do they show the correct values?
- Consider changing view objects that have nested queries to fetch lookup data to use reference entity objects instead. This can increase performance significantly!
- Create additional view objects, view accessors and list of values for dropdown lists in the original form: dropdown lists are always populated through PL/SQL code that performs some query, and are not migrated over because PL/SQL code is not parsed.

14.5.2. Checking and Enhancing the JHeadstart Service Definition

- Add item prompts where needed.
- Set column sortable where needed
- Simplify group-regions structure if possible. Can group regions be removed? Can intermediate region containers be removed Can table overflow setting be used to accomplish same layout with easier structure?
- Modify group-regions structure if generated layout differs from the original form in an undesirable manner.
- Consider removing groups based on a control block, useful items in such a block can often be dragged to databound groups. Control block groups used for searching can often be replaced with standard quick search or advanced search functionality.
- Enable stretching on groups and region containers if needed.
- Remove all unbound items that are not displayed. If the group containing the items no longer has any items, remove the group. If the group has detail groups, then have the first detail group take the location of the unbound group you are about to remove.

- For displayed unbound items that are populated through SQL statements in PL/SQL logic (for example in POST-QUERY or POST-CHANGE triggers): If possible, modify the SQL query of the base view object of the group and (outer) join it with the table that contains the column value that should be displayed in the unbound item. If you can successfully add this to your query (test the query in the application module tester!), you can change the unbound item to be databound and set the **Attribute Name** property to the attribute that you added to the view object by extending the query.
- Go over the items with **Display Type** `commandButton` and determine whether they are still needed. JHeadstart generates a number of standard buttons that might provide the same functionality like navigating from a master page to a detail page. For buttons that are still needed, see chapter 6, section 6.10 "Generating a Button Item" for more information on how you can add functionality to the button.
- Consider to replace groups migrated from control blocks that act as a search area in the Oracle form with standard ADF model-base quick search and/or advanced search on the actual databound group, or if that doesn't suit your needs, with a JHeadstart quick or advanced search area.
- In form layouts, all items that appear in one line in the original form, also appear on one line in JHeadstart-generated ADF page. This is accomplished using the **Display at Right of Item** property. Consider whether you want to simplify/clean up the layout using the **Columns** property on the group or item region to get nicely aligned columns with items.
- If the generated layout is OK, click the **Synchronize** button in the toolbar of the JHeadstart Application Definition Editor afterwards to feed back the new prompts as UI Hints on the underlying view object.
- **Attention:** if you make a non-displayed item after the migration displayed in the JHeadstart application Definition Editor, you should also click the **Synchronize** button for this group, to change the UI Display Hint in the view object accordingly. If you don't do this, you will get a `PropertyNotFound` exception at runtime (ADF issue)

14.5.3. 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 “Navigating Context-Sensitive to a group task Flow (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 6.13 “Conditionally Dependent Items” for more information.
- Calls to PL/SQL stored in the database. JHeadstart contains a convenience class which makes it very easy to call (packaged) procedures and functions in the database. See chapter 6 “Generating User Interface Widgets”, section 6.10.4. “Calling a PL/SQL Procedure or Function from a Button” for more information. (This section is also useful when the database PL/SQL logic should not be called from a button.)
- 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 chapter 3, section “Implementing Business Rules”.

CHAPTER

15

OraFormsFaces Generator

The JHeadstart OraFormsFaces Generator (OFFG) allows you to quickly embed many Oracle Forms in your ADF Faces web application using the OraFormsFaces™ product.

This chapter explains what OraFormsFaces is, and how JHeadstart integrates with OraFormsFaces.

15.1. Introduction into OraFormsFaces™

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 and OraFormsFaces work very well together when you want to transform existing Oracle Forms applications to ADF.

JHeadstart and OraFormsFaces can be a great combination for a gradual migration from Oracle Forms to ADF Faces. You can start of with a JHeadstart generated application that largely consists of existing Oracle Forms modules using OraFormsFaces. Then gradually you can replace individual pages that embed Oracle Forms with true ADF Faces pages than can just as easily be configured and generated from JHeadstart. You can either use the JHeadstart Forms2ADF generator to reuse the Oracle Forms definitions to create the new ADF pages, or just build the ADF pages from scratch by manually defining ADF Business Components and associated metadata in the JHeadstart Application Definition editor.

JHeadstart integrates with OraFormsFaces through the item **Display Type** property "oraFormsFaces". In chapter 6, section "Embedding Oracle Forms in JSF Pages" we explain how you can define an item in the JHeadstart Application Definition Editor that uses this display type.

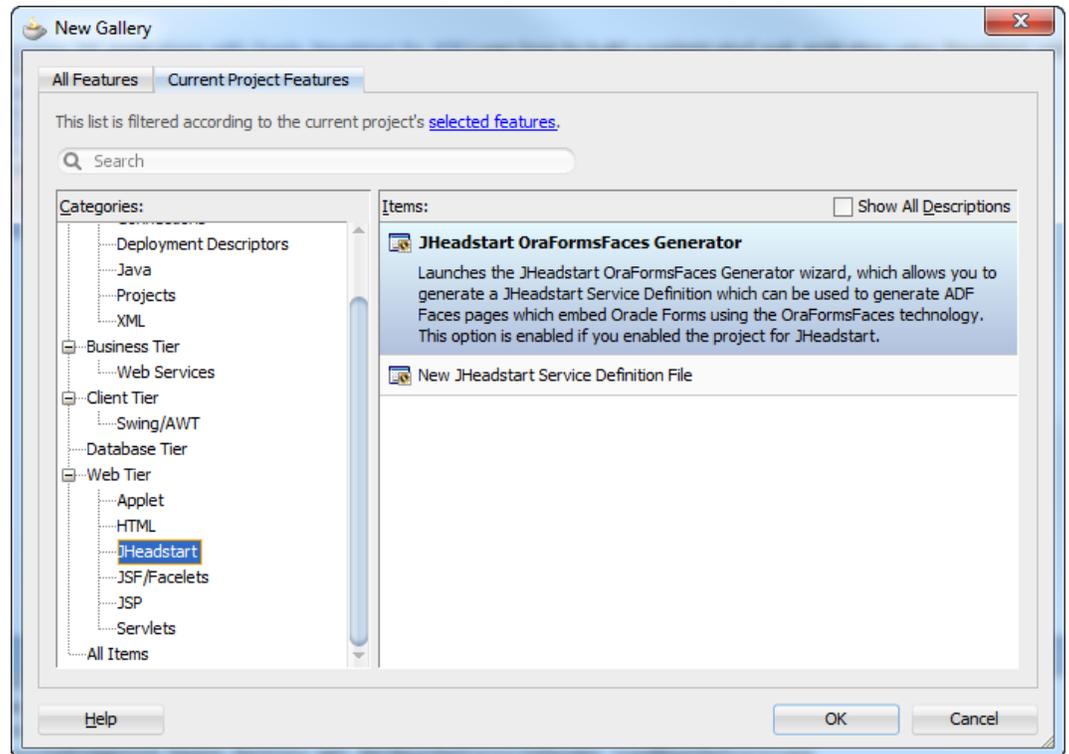
15.1.1. Added Value of the JHeadstart OraFormsFaces Generator

The OFFG simply automates the creation of a group and item with display type "oraFormsFaces" in a JHeadstart Service Definition. When running the OFFG, you can select multiple Oracle Forms, and for each selected form a group and oraFormsFaces item is created, including group parameters for each form parameter found in the form definition. These group parameters can be used if you want to deeplink from an ADF page to an embedded Oracle Form and query the context in the form.

So, you don't per se need the OFFG to embed Oracle Forms in your JHeadstart application, it just makes your life easier. In particular if you have a large Oracle Forms application with hundreds of forms, it can be a big time saver to automatically create the metadata required by JHeadstart to generate ADF pages with Oracle Forms embedded.

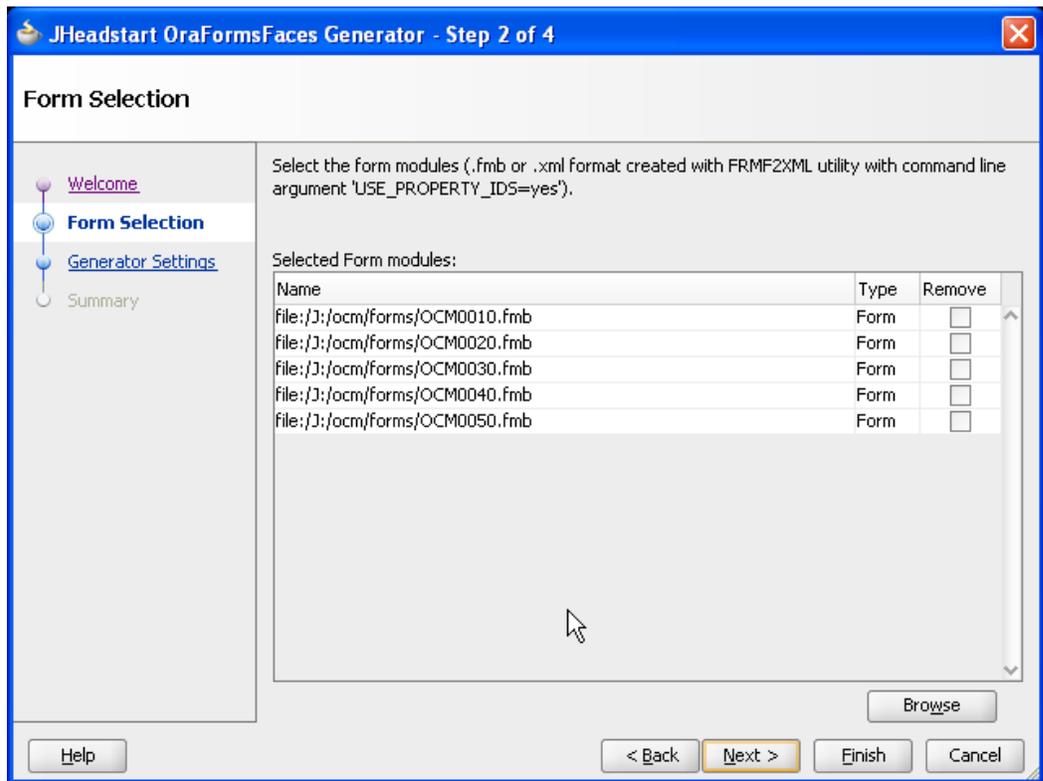
15.2. Running the JHeadstart OraFormsFaces Generator (OFFG)

You should start the OFFG from the ViewController project. To start it select your ViewController project in JDeveloper, right-mouse click, select New (or from the Menu, select File -> New), go to the JHeadstart node below the Client Tier. Select JHeadstart OraFormsFaces Generator.



15.2.1. 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.

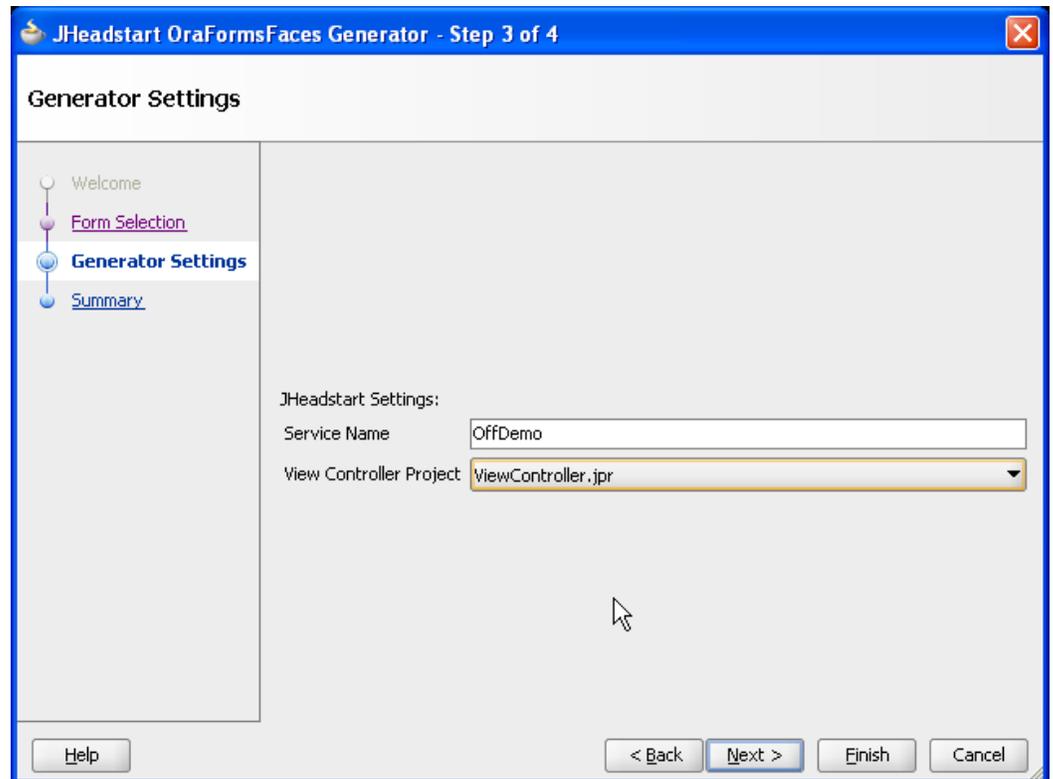


You can use the Browse button to open a File Chooser window to easily select the .fmb or .xml files from your file system.

If you select Forms .fmb files, then the OFFG 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.

15.2.2. Generator Settings

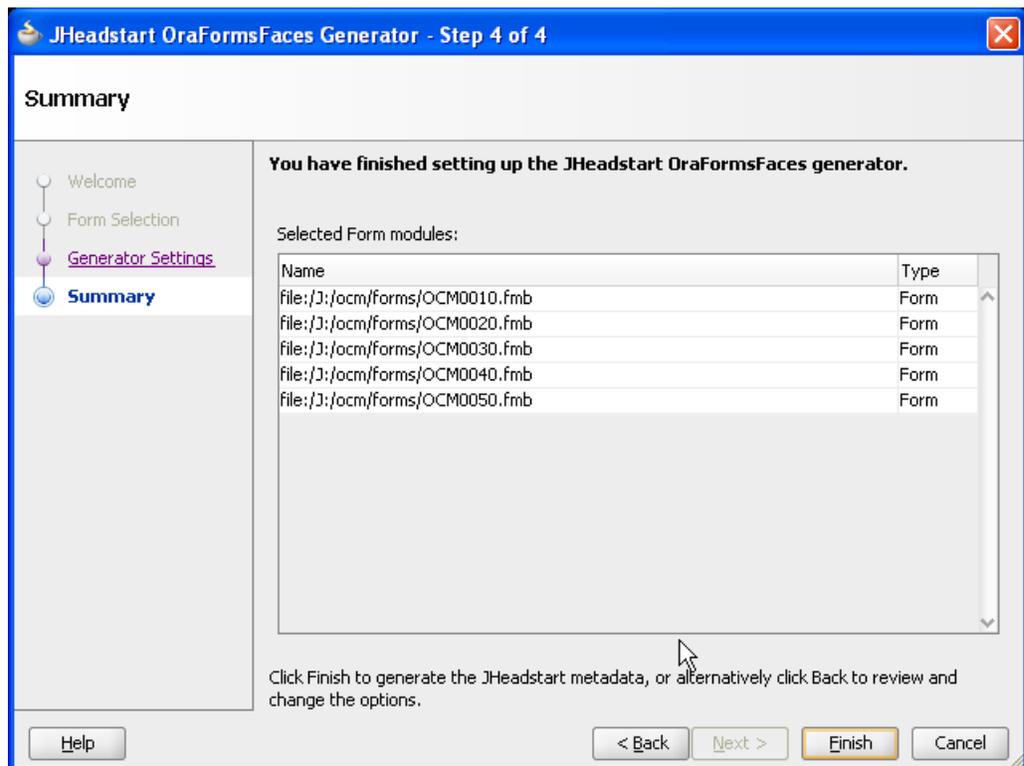


The following properties need to be set on the Generator Settings panel:

- **Service Name:** This is the name that is used to create the JHeadstart Application Definition file
- **View Controller Project:** The name of the project in which the JHeadstart Application Definition file will be saved.

15.2.3. Processing the Selected Forms

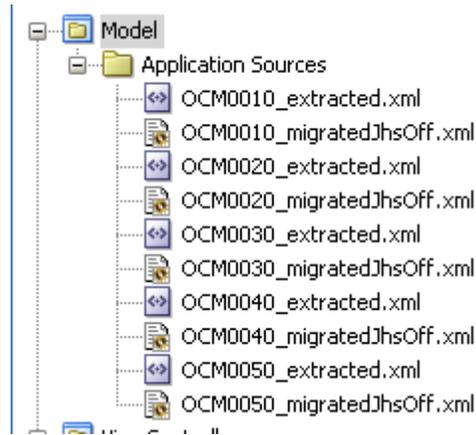
When you click the Finish button on the Summary panel, the OFFG will start processing the forms.



The processing consists of the following steps:

- **Extraction:** 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.
- **JHeadstart Migration:** Each 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 OFFG run was successful, you can safely delete the intermediate results of the OFFG, 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.



15.2.4. Troubleshooting

When the OFFG 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 shown in the screen shot below.

```
JHeadstart OraFormsFaces Generator - Log
INFORMATION: Extracting file:/J:/ocm/forms/OCM0010.fmb
INFORMATION: Extracting file:/J:/ocm/forms/OCM0020.fmb
INFORMATION: Extracting file:/J:/ocm/forms/OCM0030.fmb
INFORMATION: Extracting file:/J:/ocm/forms/OCM0040.fmb
INFORMATION: Extracting file:/J:/ocm/forms/OCM0050.fmb
INFORMATION: Creating JhsOff migration XML for OCM0010
INFORMATION: Creating JhsOff migration XML for OCM0020
INFORMATION: Creating JhsOff migration XML for OCM0030
INFORMATION: Creating JhsOff migration XML for OCM0040
INFORMATION: Creating JhsOff migration XML for OCM0050
INFORMATION: Composing JHeadstart Application Definition for OCM0010
INFORMATION: Composing JHeadstart Application Definition for OCM0020
INFORMATION: Composing JHeadstart Application Definition for OCM0030
INFORMATION: Composing JHeadstart Application Definition for OCM0040
INFORMATION: Composing JHeadstart Application Definition for OCM0050
```

The last printed line in this window will tell you which forms module the OFFG was processing when the error occurred, and in which phase of the processing.

In such a situation, we recommend that you e-mail the Oracle JHeadstart Team (idevcoe_nl@oracle.com), and send us the following information:

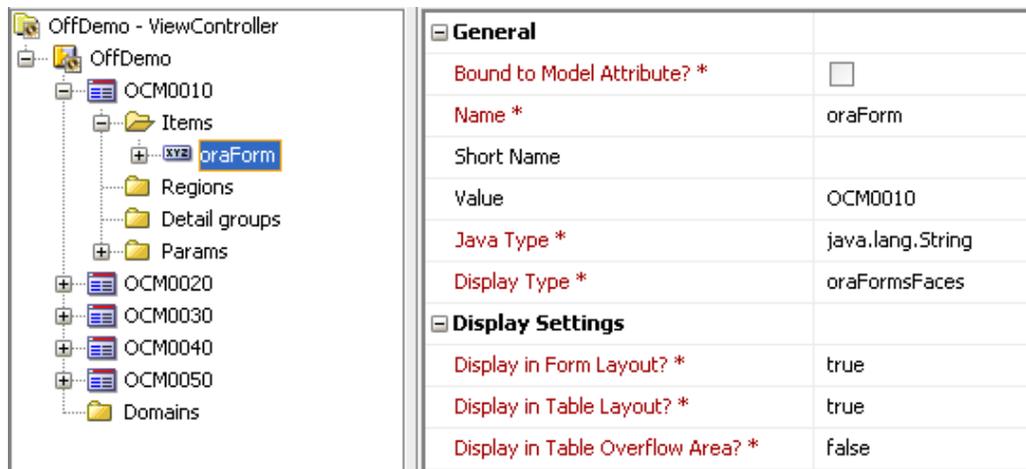
- The JHeadstart version you are using. You can see this in JDeveloper by going to the Help menu -> JHeadstart Documentation Index.
- The name of the form module that is causing the problem.
- The error message and error stack trace displayed in the dialog window (if any)
- Any log information written to the JDeveloper OFFG log window
- The XML files created during processing for this module: `_extracted.xml`, and `migratedJhsOff.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.

15.3. Understanding the Outputs of the JHeadstart OraFormsFaces Generator

The JFG creates a JHeadstart Service Definition file. To see and edit this file using the JHeadstart Application Definition Editor, you first need to enable JHeadstart for the ViewController project if you didn't do this yet.

For each selected form module the following content is created:

- A top-level **Group** is created
- For each Forms parameter, a **Group Parameter** is created
- An item with **Display Type** oraFormsFaces is created in the group with the **Value** property set to the name of the form.
- For each Forms parameter, an **Item Parameter** is created, with the value set to the EL expression that references the group parameter (generated as taskflow parameter by JHeadstart): `{pageFlowScope.parameterName}`



The screenshot shows the JHeadstart Application Definition Editor. On the left, a tree view displays the project structure: OffDemo - ViewController > OffDemo > OCM0010 > Items > oraForm. The 'oraForm' item is selected. On the right, the 'General' and 'Display Settings' tabs are visible. The 'General' tab shows the following configuration:

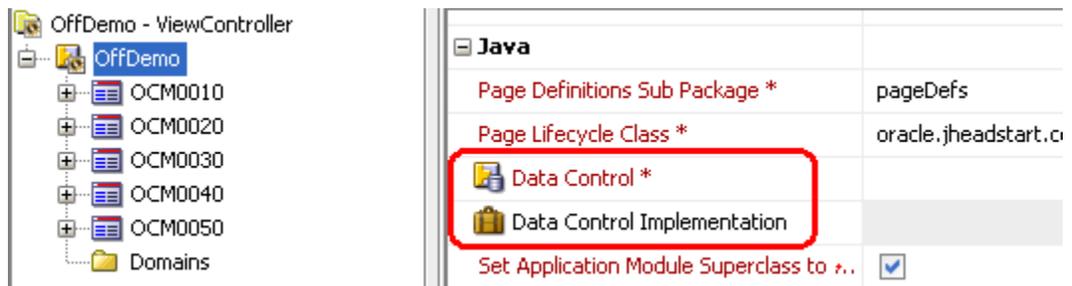
General	
Bound to Model Attribute? *	<input type="checkbox"/>
Name *	oraForm
Short Name	
Value	OCM0010
Java Type *	java.lang.String
Display Type *	oraFormsFaces

The 'Display Settings' tab shows the following configuration:

Display Settings	
Display in Form Layout? *	true
Display in Table Layout? *	true
Display in Table Overflow Area? *	false

15.3.1. Create Application Module If Needed

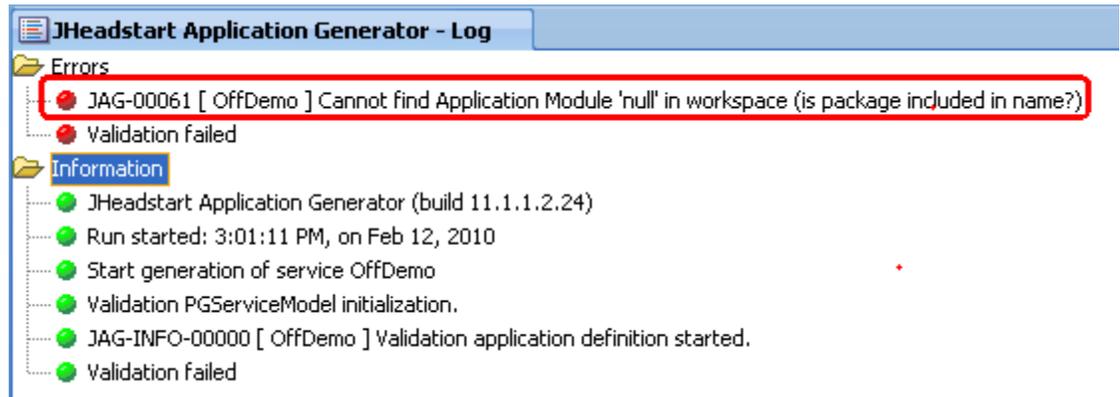
If you have run the OFFG in a new project without an existing application module, then the Data Control property is not set in the generated service definition.



The screenshot shows the JHeadstart Application Definition Editor. On the left, a tree view displays the project structure: OffDemo - ViewController > OffDemo > OCM0010. The 'OffDemo' application module is selected. On the right, the 'Java' tab is visible, showing the following configuration:

Java	
Page Definitions Sub Package *	pageDefs
Page Lifecycle Class *	oracle.jheadstart.c
Data Control *	
Data Control Implementation	
Set Application Module Superclass to r..	<input checked="" type="checkbox"/>

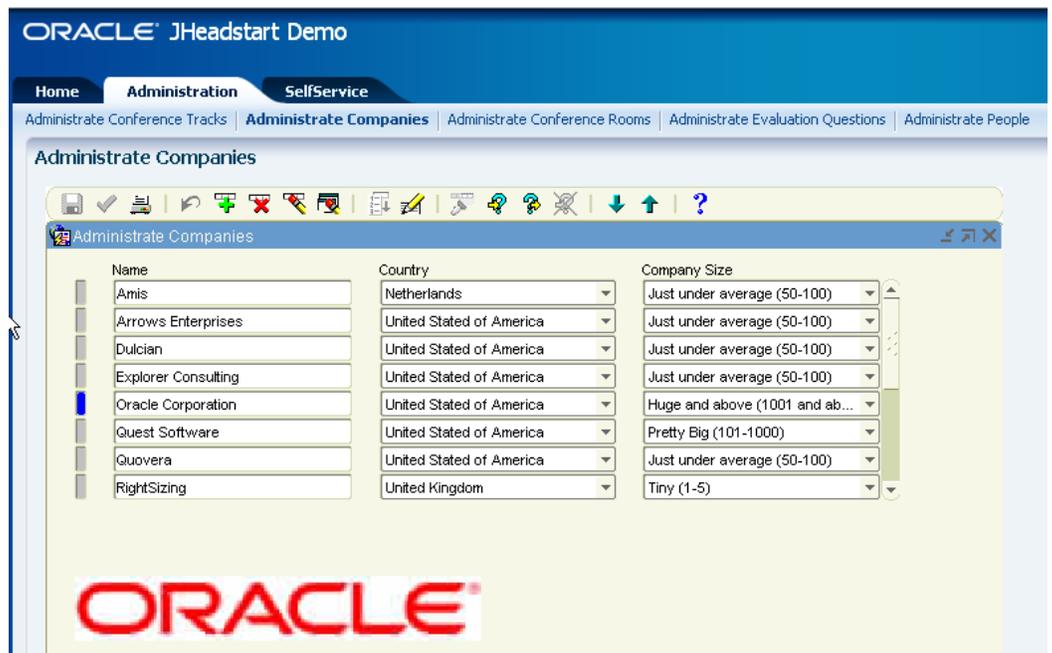
If you then try to run the JHeadstart Application Generator, you will get the following error:



To solve this error, you need to create a “dummy” application module in your Model project and then set the Data Control property to the data control automatically created for this application module. Note that the data model of the application module can be empty, it doesn't need to contain a view object usage.

You then can generate and run your application. Make sure you have properly installed OraFormsFaces in your project before you run the application.

If you experience any problems, then first manually create and test an ADF Faces page that uses the OraFormsFaces JSF components to make sure OraFormsFaces is configured correctly you in your project. See the OraFormsFaces documentation for more info.



This page is intentionally left blank.