Professional Expertise Distilled

Oracle ADF Enterprise Application Development—Made Simple

Successfully plan, develop, test, and deploy enterprise applications with Oracle ADF

Sten E. Vesterli

[PACKT] PUBLISHING enterprise professional expertise distilled

Oracle ADF Enterprise Application Development—Made Simple

Successfully plan, develop, test, and deploy enterprise applications with Oracle ADF

Sten E. Vesterli

[PACKT] PUBLISHING enterprise
professional expertise distilled

BIRMINGHAM - MUMBAI

Oracle ADF Enterprise Application Development—Made Simple

First published: June 2011

Production Reference: 1060611

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849681-88-9

`www.packtpub.com`

Cover Image by David Guettirrez (`bilbaorocker@yahoo.co.uk`)

Credits

Author
Sten E. Vesterli

Reviewers
Aino Andriessen
Duncan Mills
Frank Nimphius
Grant Ronald

Acquisition Editors
Dhwani Devater
Rashmi Phadnis

Development Editor
Hyacintha D'Souza

Technical Editor
Aditi Suvarna

Project Coordinator
Vishal Bodwani

Proofreader
Stephen Swaney

Indexer
Rekha Nair

Graphics
Geetanjali Sawant

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

Sten E. Vesterli picked up Oracle development as his first job after graduating from the Technical University of Denmark, and hasn't looked back since. He has worked with almost every development tool and server Oracle has produced in the last decade and a half, including Oracle ADF, JDeveloper, WebLogic, SQL Developer, Portal, BPEL, Collaboration Suite, Designer, Forms, Reports, and even Oracle Power Objects.

He started sharing his knowledge with a conference presentation in 1997 and has since given more than 50 conference presentations at Oracle OpenWorld and at ODTUG, IOUG, UKOUG, DOAG, and other user group conferences. His presentations are consistently highly rated by the participants, and in 2010 he received the ODTUG Best Speaker award.

He has also written numerous articles, participated in podcasts, and written the book Oracle Web Applications 101.

Oracle has recognized Sten's skills as an expert communicator on Oracle technology by awarding him the prestigious title Oracle ACE Director, carried by less than 100 people in the world. He is also an Oracle Fusion User Experience Advocate and sits on the Oracle Usability Advisory Board.

Based in Denmark, Sten is a partner in the Oracle consulting company Scott/Tiger, where he works as a senior principal consultant. When not writing books or presenting, he is helping customers choose the appropriate technology for their needs, teaching, mentoring, and leading development projects.

Acknowledgement

I'd like to thank the many members of the ADF Enterprise Methodology Group (ADF EMG) who meet online and in person to discuss and share the best practices. Your insights have helped shape my opinion on good enterprise ADF development and have improved the book. In particular, I'd like to thank the group founder Chris Muir, as well as moderators and organizers Simon Haslam and John Flack, who ensure that the discussions on the group mailing list stay on topic. If you are serious about enterprise ADF development, you need to join this group: `http://groups.google.com/group/adf-methodology`.

Other ADF EMG members I'd like to single out for special mention include John Stegeman (author of the ADF Essentials series on Oracle Technology Network), prolific ADF blogger Andrejus Baranovskis, enterprise developer Aino Andriessen.

Oracle's Laura Akel, Susan Duncan, Duncan Mills, Frank Nimphius, and Grant Ronald have also provided valuable information, feedback, and access to the latest software - I am grateful for the time you have taken to comment on this book and show me new features.

I'd also like to thank my children, Michael and Maria, for patiently waiting for my Tiefling Rogue to return to our gaming sessions to continue the battle against the undead, and for learning to make crêpes when daddy didn't have time.

And finally, I'd like to thank my wonderful wife Lotte for her unhesitating support for the idea of me writing another book, for taking care of my tasks in the household while I was writing, and for our coffee breaks together when I needed to recharge my batteries.

About the Reviewers

Aino Andriessen is a principal consultant and expertise lead in Application Lifecycle Management at AMIS; an Oracle, Java, and SOA specialist based in the Netherlands. His focus is on Oracle ADF, JHeadstart, SOA, and Enterprise Java development, application lifecycle management, architecture, and quality management. He is a frequent presenter at the ODTUG Kaleidoscope, Oracle Open World, and UKOUG TechEBS. He writes articles and publishes them at the AMIS technology blog (`http://technology.amis.nl/blog/`).

Duncan Mills is a senior director of product management for Oracle's Application Development Tools including Oracle JDeveloper, Oracle Enterprise Pack for Eclipse, NetBeans, Oracle Forms, and the ADF Framework. Duncan is currently responsible for product direction, evangelism, and courseware development around the development tools products. He has been working with Oracle in a variety of application development and DBA roles since 1988. For the past nineteen years he has been working at Oracle in both support and product development, spending the last eight years in product management. Duncan is the co-author of the Oracle Press books Oracle JDeveloper 10g for Forms and PL/SQL Developers: A Guide to Web Development with Oracle ADF and Oracle JDeveloper 11g Handbook: A Guide to Fusion Web Development.

Frank Nimphius is a senior principal product manager in the Application Development Tools organization at Oracle Corporation. In his product management role, Frank contributes to the development and the evangelization of the Oracle JDeveloper and Oracle Application Development Framework (ADF) products. Frank runs the ADF Code Corner website (`http://www.oracle.com/technetwork/developer-tools/adf/learnmore/index-101235.html`) and publishes on the OTN Harvest blog (`http://blogs.oracle.com/jdevotnharvest/`). He is the co-author of the Oracle Fusion Developer Guide book, published by Oracle Press in 2010.

Grant Ronald is a senior group product manager working for Oracle's Application Development Tools group responsible for Forms and JDeveloper where he has a focus on opening up the Java platform to Oracle's current install base. Grant joined Oracle in 1997, working in Oracle support, where he headed up the Forms/Reports/Discoverer team responsible for the support of the local Oracle Support Centers throughout Europe, Middle East, and Africa. Prior to Oracle, Grant worked for seven years in various development roles at EDS Defence.

Grant is author of the Quick Start Guide to Oracle Fusion Development: Oracle JDeveloper and Oracle ADF, published in 2010.

To Lotte, Michael, and Maria

PACKT
PUBLISHING

Table of Contents

Preface

Welcome to your first real-life enterprise ADF application!

The book you are holding in your hands is about building serious applications with Oracle Application Development Framework (ADF). You know that actual development work is only one part of a successful project, and that you also need structure, processes, and tools.

That is why this book will take an *enterprise* focus, following a complete project from inception to final delivery. Along the way, you will be building a proof of concept application, but you will also be setting up and using all the professional support tools you need for a real-life project.

This book will take you through the entire process of building an enterprise ADF application – from the initial idea through proof of concept, tool choice, preparation, coding support classes, building the application, testing it, customizing it, securing it, and finally deploying it.

What is an enterprise application?

Enterprise applications are the strategic applications in the enterprise. They will handle critical business functions and tend to be big and complex. In the past, it was acceptable that users had to take training classes before they were able to use the application, but today, enterprise applications are also required to be user-friendly and intuitive. As they are deployed throughout the organization, they will need sophisticated security features. And because of the cost of developing and implementing enterprise applications, they will remain in use for a long time.

Application size

An enterprise application is big – containing lots and lots of code modules, interacting in complex ways among themselves and with external systems.

Typically, this means that an enterprise application also has a lot of different screens where the user will interact with the system. However, it is also possible that the complexity of the enterprise application is hidden from the user; a good enterprise application might seem deceptively simple to the average user.

Development team

The complexity of an enterprise application means that it will have to be built by a larger team. It will use several technologies, so you need people skilled in all the relevant areas. And because of its sheer size, you will need to have people working in parallel on different parts of the application in order to develop it within a useful timeframe.

Because of the interdependencies among the different parts of the application, an enterprise application cannot simply be partitioned out among developers. Instead, development work must be carefully planned so that the foundation is laid down before the rest of the house is built – while at the same time allowing for the inevitable changes as the project progresses.

Development tools

Naturally, you need an integrated development environment (IDE) to build the actual application. This book assumes that the entire team will be using Oracle's free JDeveloper tool for all work. The choice of IDE can be the subject of almost religious fervor and some projects allow each developer to choose his or her favorite IDE. However, in an enterprise project, the benefits from having everyone use the same tool clearly outweighs any minor benefit achieved by using other IDEs with marginally better support for one or the other task.

In addition to the IDE, you will also need source control – a server holding all the different versions of the development artifacts, and a client on each development workstation. This book uses the popular Subversion tool as an example of how to use source control in an enterprise project with JDeveloper.

Another important tool is an issue-tracking tool. This can be used to track defects in code as well as ideas, development tasks, and many other things. In this book, the well-known Jira tool is used, integrated into Oracle Team Productivity Center (TPC). The use of TPC allows the development team to link Jira issues with code artifacts for maximum traceability.

Finally, you need a scripting tool. In a small project, it might be sufficient to build applications directly off the IDE, but in an enterprise application, you need a tool to ensure that you can build your project in a consistent manner. This book uses Ant as an example of a scripting tool for ADF projects.

Lifetime of an enterprise application

Because of the effort and cost involved in building enterprise applications, they are not casually thrown away and re-built. Indeed, many organizations are still running enterprise applications built more than a decade ago.

The longevity of enterprise applications makes it extremely important that they are well built and well documented. Most developers will be familiar with the pain of having to maintain a poorly documented application, and understand the need for good documentation.

But while documentation is important, it is just as important that the application is built in a recognizable, standard way. This is why this book advocates using the ADF framework in its intended way – so that coming generations of developers can look at the code and immediately understand how the application is built.

What this book covers

Before your organization embarks on building an enterprise application using Oracle Application Development Framework, you need to prove that ADF will indeed be able to meet the application requirements.

Chapter 1, *The ADF Proof of Concept*, will take you through building a proof of concept application using the normal ADF components: ADF Business Components for the middle tier and ADF Faces and ADF Task Flows for the user interface. The application will access data stored in relational tables and use both the standard ADF components and an ADF Data Visualization component (a Gantt chart). This chapter contains step-by-step instructions and can be used as a hands-on exercise in basic ADF development.

Once you have proved that ADF is capable of delivering the necessary functionality, you need to figure out which components will be part of your application, and to estimate the total effort necessary to build it.

Chapter 2, *Estimating the Effort*, will provide checklists of task you must remember in your estimate as well as some guidelines and estimation techniques that you can use to calculate how much time it will take to build the application.

The next step after having decided to proceed with an ADF enterprise project is to organize the development team.

Chapter 3, *Getting Organized*, explains the skill you need to build an enterprise application, and how to organize your team. It also explains which tools you need in your enterprise project, and how you should structure your code using separate workspaces connected through the powerful ADF Library functionality for maximum efficiency.

In order for the team to work efficiently towards the project goal, each developer needs a development workstation with full integration to all necessary tools.

Chapter 4, *Productive Teamwork*, describes how to set up and use Oracle Team Productivity Center, which serves as an integration hub, connecting your issue tracking system (for example, Jira) and other tools to JDeveloper. It also explains how to work effectively with Subversion and JDeveloper together for version control.

With your workstation all set up and ready to go, you need one more thing before starting development in earnest: Templates and framework extension classes. For a small application it might be OK to just start coding and work out the details as you go along. However, in an enterprise application, the rework cost of such an informal approach can be prohibitive.

Chapter 5, *Prepare to Build*, explains the task flow and page templates you need to build a uniform user interface in an efficient way, explains why you need your own ADF framework extension classes, and how to build these.

Now that all the infrastructure and templates are in place, and the development workstation has been configured with all necessary connections, it is time to prove the entire development flow.

Chapter 6, *Building the Enterprise Application*, walks you through creating the same proof of concept application as in Chapter 1, but this time using all the enterprise support tools configured and prepared in Chapters 4 and 5. The application is built in a module manner in separate subsystems and integrated together in a master application to illustrate how a large enterprise application should be structured.

By the end of this chapter, you will have proved that the entire enterprise toolset is functional and have re-built the proof of concept application using correct enterprise methodology.

All applications need to be tested – but enterprise applications need testing much more than smaller applications for two reasons:

- The size and complexity of an enterprise application means that there are more interfaces where things can go wrong
- The long expected life of an enterprise application makes it almost certain that other developers will be maintaining it in years to come

For both of these reasons, it is important that the application comes with test cases that prove correct functionality. It will not be sufficient to have a collection of test scripts that must be manually executed – these will not be consistently executed and will surely become out of date over the lifetime of the application. Your tests must therefore be automated so they can be executed as part of the build process.

Chapter 7, *Testing your Application*, explains how to write code tests in the form of JUnit test cases, how to use Selenium to record and play back user interface tests, and how to use JMeter to for load testing your ADF application.

Your organization will, of course, have graphical standards that the application must adhere to. In an ADF application, the look of the application can easily be modified in a process known as "skinning". By developing several skins, you can even deploy the same application multiple times with very different visual identities – an invaluable feature for independent software vendors.

Chapter 8, *Look and Feel*, explains how to use the powerful skin editor that is new in JDeveloper 11g release 2 to create Cascading Style Sheets to create a new "skin" for your application corresponding to your enterprise visual identity.

Looking at the requirements for your application, you might identify a number of pages or screens that are almost, but not quite, identical. In many cases, you do not have to develop each of these individually – you might be able to develop one master page and use functional customization to provide different groups of users with different versions of the page.

The ability to easily customize application functionality is one of the truly outstanding features of the Oracle ADF framework. Here, you benefit from the fact that Oracle has developed ADF for real-life, large enterprise applications like Oracle Fusion Applications. And, if you are an independent software vendor producing software for sale, you can use this feature to easily customize a base application for individual customers.

Chapter 9, *Customizing the Functionality*, explains how to set up an application for customization using ADF Meta Data Services and how to use the special JDeveloper "customization" role to perform the actual customization.

Your enterprise application needs a robust, role-based security model.

Chapter 10, *Securing your ADF Application*, explains how to secure both user interface (task flows and pages) and data access (Entity Objects) using ADF security features, and how ADF security ties in with WebLogic security features.

Once the individual parts of the application have been built and tested, it is time to build a complete deployment package.

Chapter 11, *Package and Deliver*, describes how an enterprise application deployment package is built, and how the development team can set up their own stand-alone WebLogic server to ensure that the deployment package will work when handed over to the operations team.

An enterprise application might have to be made available in several languages.

Appendix A, *Internationalization*, explains how internationalization works in ADF, and how to produce a localized application.

How to read this book

This book follows an enterprise application from inception to final delivery, and you can read the chapters in sequence to learn a proven method for successfully building an enterprise application that meets business requirements, on time and on budget.

However, each chapter can also be read on its own if you just need information on a specific topic. For example:

- *Chapter 1*, *The ADF Proof of Concept*, can serve as a quick introduction to ADF, allowing an experienced developer to get started with ADF
- *Chapter 4*, *Productive Teamwork*, explains how to set up and use Oracle Team Productivity Center to integrate issue tracking with Jira into JDeveloper
- *Chapter 9*, *Customizing the Functionality*, explains how to use the power of Meta Data Services to build a customizable ADF application

Ready to build a real-life enterprise application? Let's get started!

What you need for this book

- **Oracle JDeveloper**: This essential tool is free and can be downloaded from the Oracle Technology Network (`http://otn.oracle.com`). The examples in this book use version 11.1.1.4, but you should easily be able to use 11.1.1.5 or JDeveloper 11g Release 2.
- **A database**: You can use the free Oracle Database Express Edition 11g Release 2, which is also available from the Oracle Technology Network.
- **Version control software**: This book uses Subversion as an example, but there are many other fine version control systems available.
- **Issue tracking software**: This book uses Jira from Atlassian, but many other options are available.
- **A scripting tool**: This book uses and recommends Apache Ant.

Who this book is for

This book is for developers in general - both web developers and developers experienced with classic 4GL tools such as Oracle Forms - who wish to learn how to develop modern, user-friendly web applications in an Oracle environment. It is for novice ADF developers who wish to learn how to use JDeveloper and ADF, as well as for more experienced ADF developers who wish to improve their knowledge and understanding of ADF and how to use it effectively.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Choose the `ElemKey` attribute and then click on the green plus sign."

A block of code is set as follows:

```
(:pResponsible is null or PERS_ID = :pResponsible)
and (:pProgramme is null or PROG_ID = :pProgramme)
and (:pText is null or upper(TEXT) like '%' || upper(:pText) || '%')
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
public class RoleLayerCC extends CustomizationClass {
public CacheHint getCacheHint() {
return CacheHint.MULTI_USER;
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "When you are done entering the WHERE clause, click on the **Test** button to verify that your SQL is valid."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1 The ADF Proof of Concept

Your organization has decided that ADF might be the right tool to build your next enterprise application—now you need to set up an experiment to prove that your assumption is correct.

You can compare the situation at the start of a project to standing in front of a mountain with the task to excavate a tunnel. The mountainsides are almost vertical, and there is no way for you to climb the mountain to figure out how wide it is. You can take two approaches:

- You can either start blasting and drilling in the full width of the tunnel you need
- You can start drilling a very small pilot tunnel all through the mountain, and then expand it to full width later

It's probably more efficient to build in the full width of the tunnel straight from the beginning, but this approach has some serious disadvantages as well. You don't know how wide the mountain is, so you can't tell how long it will take to build the tunnel. In addition, you don't know what kind of surprises might lurk in the mountain—porous rock, aquifers, or any number of other obstacles to your tunnel building.

That's why you should build the pilot tunnel first—so you know the size of the task and have an idea of the obstacles you might meet on the way.

The **Proof of Concept** is that pilot tunnel.

The very brief ADF primer

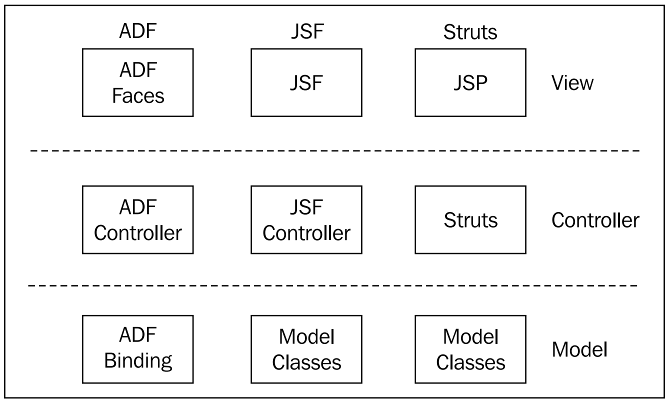
Since you have decided to evaluate ADF for your enterprise application, you probably already have a pretty good idea of its architecture and capabilities. Therefore, this section will only give a very brief overview of ADF—there are many whitepapers, tutorials, and demonstrations available at the Oracle Technology Network website. Your starting point for ADF information is `http://otn.oracle.com/developer-tools/jdev/overview`.

Enterprise architecture

A modern enterprise application typically consists of a frontend, user-facing part and a backend business service part.

Frontend

The frontend part is constructed from several layers. In a web-based application, these are normally arranged in the common **Model-View-Controller** (**MVC**) pattern as illustrated next:



The **View** layer is interacting with the user, displaying data as well as receiving updates and user actions. The **Controller** layer is in charge of interpreting user actions and deciding which screens are presented to the user in which order. And the **Model** layer is representing the backend business services to the **View** and **Controller**, hiding the complexity of storing and retrieving data.

This architecture implements a clean separation of duties— the page doesn't have to worry about where to go next, because that is the task of the controller. And the controller doesn't have to worry about how to store data in the data service, because that is the task of the model.

Other Frontends

An enterprise application could also have a desktop application frontend, and might have additional frontends for mobile users or even use existing desktop applications like Microsoft Excel to interact with data. In the ADF technology stack, all of these alternative frontends interact with the same model, making it easy to develop multiple frontend applications against the same data services.

Backend

The backend part consists of a **business service** layer that implements the business logic and provide some way of accessing the underlying data services. Business services can be implemented as API code written in Java, PL/SQL or other languages, web services, or using a business service framework such as ADF Business Components.

Under the business services layer there will be a **data service** layer actually storing persistent data. Typically, this is based on relational tables, but it could also be XML files in a file system or data in other systems accessed through an interface.

ADF architecture

There are many different ways of building applications with Oracle Application Development Framework, but Oracle has chosen a modern SOA-based architecture for Oracle Fusion Applications. This brand new product has been built from the ground up as the successor to Oracle E-Business Suite, Siebel, PeopleSoft, J.D. Edwards and many other applications Oracle has acquired over the last couple of years.

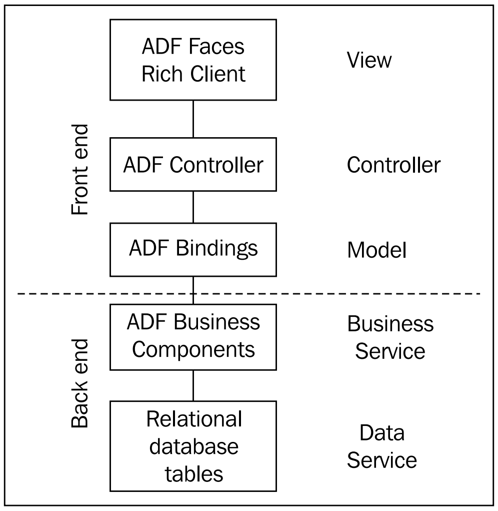
If it is good enough for Oracle Fusion Applications, arguably the biggest enterprise application development effort ever undertaken by mankind, it is probably good enough for you, too.

Oracle Fusion Applications are using the following parts of the ADF framework:

- **ADF Faces Rich Client** (**ADFv**), a very rich set of user interface components implementing advanced functionality in a web application.
- **ADF Controller** (**ADFc**), implementing the features of a normal JSF controller, but extended with the possibility to define modular, reusable page flows. ADFc also allows you to declare transaction boundaries so one database transaction can span many pages.
- **ADF binding layer** (**ADFm**), standard defining a common backend model that the user interface can communicate with.

- **ADF Business Components** (**ADFbc**), a highly productive, declarative way of defining business services based on relational tables.

You can see all of these in the following figure:



There are many ways of getting from A to B—this book is about travelling the straight and well-paved road Oracle has built for Fusion Applications. However, other routes might be appropriate in some situations: You could build the user interface as a desktop application using ADF Swing components, you could use ADF for a mobile device, or you could use ADF Desktop Integration to access your data directly from within Microsoft Excel. Your business services could be based on Web Services, EJBs or many other technologies, using the ADF binding layer to connect to the user interface.

Entity objects and associations

Entity objects (**EOs**) takes care of object-relational mapping: Making your relational tables available to the application as Java objects. Entity objects are the base that view objects are built on, and all data modifications go through the entity object. You will normally have one entity object for every database table or database view your application uses, and this object is responsible for producing the correct SQL statements to insert, update or delete in the underlying relational tables.

The entity objects helps you build scalable and well-performing applications by intelligently caching records on the application server in order to minimize the load the application places on the database.

Like entity objects are the middle-tier reflection of database tables and database views, **Associations** are the reflection of foreign key relationships between tables. An association represents a connection between two entity objects and allows ADF to relate data in one entity object with data in another. JDeveloper is normally able to create these automatically by simply inspecting the database, but in case your database does not contain foreign keys, you can build associations by hand to tell ADF about the relationships in your data.

View objects and View Links

While you do not really need to make any major decisions when building the entity objects for the Proof of Concept, you do need to consider the consumers of your business services when you start building **view objects**—for example, what information you would display on a screen.

View objects are typically based on entity objects and you will be using them for two purposes:

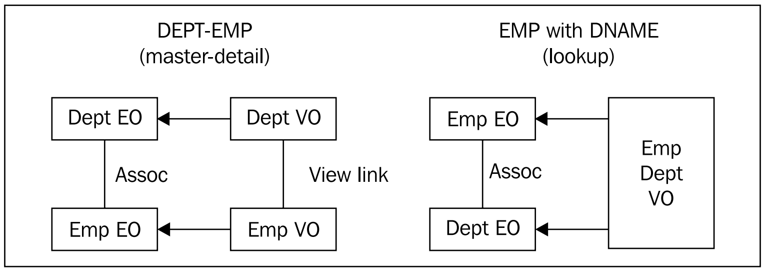
- To provide data for your screens
- To provide data for **lists of values** (**LOVs**)

The data handling view objects are normally specific for each screen or business service. One screen can use multiple view objects—in general, you need to create one view object for each master-detail level you wish to display on your screen. One view object can pull together data from several entity objects, so if you just need to retrieve a reference value from another table, you do not need to create a separate view object for this.

The LOV view objects are used for drop-down lists and other selections in your user interface. They will typically be defined as read-only and because they are reusable, you will define them once and re-use them everywhere you need a drop-down list on a specific data set.

View Links are used to define the relationships betweens the view objects and are typically based on associations (again often based on foreign keys in the database).

The following figure shows an example of two ways to display the data from the familiar EMP and DEPT tables. The left-hand illustration shows a situation where you wish to display a department with all the employees of the department in a master-detail screen. In this case, you create two view objects connected by a view link. The right-hand illustration shows a situation where you wish to display all employees, together with the name of the department where they work. In this case, you only need one view object, pulling together data from both the EMP and DEPT tables through the entity objects.



Application modules

Application modules encapsulate the view object instances and business service methods necessary to perform a unit of work. Each application module has its own transactional context and holds its own database connection. This means that all of the work a user performs using view objects from one application module is part of one database transaction.

Application modules can have different granularity, but typically, you will have one application module for each major piece of functionality. If your requirements are specified with use cases, there will often be one application module for each major use case. However, multiple use cases can also be grouped together into one application module – indeed, it is possible to build a small application using just one application modules.

Application modules for Oracle Forms

If you come from an Oracle Forms background and are developing a replacement for an Oracle Forms application, your application will often have a relatively small number of complex, major Forms, and larger number of simple data maintenance Forms. You will often create one Application Module per major Form, and a few application modules that each provide data for a number of simple Forms.

If you wish, you can combine multiple application modules inside one **root application module**. This is called **nesting** and allows several application modules to participate in the transaction of the root application module. This also saves database connections because only the root application module needs a connection.

The ADF user interface

The preferred way to build the user interface in an ADF enterprise application is with **JavaServer Faces** (**JSF**). JSF is a component-based framework for building web-based user interfaces that overcome many of the limitations of earlier technologies like **JavaServer Pages** (**JSP**).

In a JSF application, the user interface does not contain any code, but is instead built from configurable components from a component library. For your application, you will want to use the sophisticated ADF 11g JavaServer Faces (JSF) component library, known as the **ADF Faces Rich Client**.

> There are other JSF component libraries—for example, the previous version of the ADF Faces components (version 10g) has been released by Oracle as Open Source and is now part of the Apache MyFaces Trinidad project. But for a modern enterprise application, use ADF Faces Rich Client.

ADF Task Flows

One of the great improvements in ADF 11*g* was the addition of **ADF Task Flows**.

It had long been clear to web developers that in a web application, you cannot just let each page decide where to go next—you need the controller from the MVC architecture. Various frameworks and technologies have implemented controllers (both the popular Struts framework and JSF has this), but the controller in ADF Task Flows is the first controller capable of handling large enterprise applications.

An ADF web application has one **Unbounded Task Flow** where you place all the publicly accessible pages and define the navigation between them. This corresponds to other controller architectures. But ADF also has **Bounded Task Flows**, which are complete, reusable mini-applications that can be called from the unbounded task flow or from another bounded task flow.

A bounded task flow has a well-defined entry point, accepts input parameters and can deliver an outcome back to the caller. For example, you might build a customer management task flow to handle customer data. In this way, your application can be built in a modular fashion—the developers in charge of implementing each use case can define their own bounded task flow with a well-defined interface for others to call. The team building the customer management task flow is thus free to add new pages or change the navigation flow without affecting the rest of the application.

ADF pages and fragments

In your task flows, you can define either **pages** or **page fragments**. Pages are complete web pages that you can run on their own, while page fragments are reusable components that you place inside regions on pages. An enterprise application will often have a small number of pages (possibly only one), and a larger number of page fragments that dynamically replace each other inside a region. This design means that the user does not see the whole browser window redraw itself—only parts of the page will change as one fragment is replaced with another. It is this technique that makes an ADF application seem more like a desktop application than a traditional web application.

On your pages or page fragments, you add content using layout components, data components and control components:

- The layout components are containers for other components and control the screen layout. Often, multiple layout components are nested inside each other to achieve the desired layout.
- The data components are the fields, drop-down lists, radio buttons and so on that the user interacts with to create and modify data.
- The control components are the buttons and links used to perform actions in an ADF application.

The Proof of Concept

The Proof of Concept serves two purposes:

- To demonstrate that the technology works
- To gather some metrics about your development speed

If we return to the tunnel analogy, we need to demonstrate that we can drill all the way through the mountain, and measure our drilling speed.

What goes into a Proof of Concept?

The most important part of the Proof of Concept is that it goes all the way through the mountain – or in application development terms: All the way from the user interface to the backend data service and back.

If your data service is data in relational tables and you will be presenting it in ordinary fields and tables on the screen, the part of your proof of concept that demonstrates the technology is fairly straightforward.

However, if your data service is not just relational tables – if you are using Web Services or API code in C++, Java, or PL/SQL, you need to demonstrate that you can retrieve data from your data service, display it on the screen, modify it and successfully store the changes in the backend data service.

You might also have user interface requirements that require more advanced components like trees, graphs, or even drag-and-drop functionality for the end user. If that is the case, your proof of concept user interface needs to demonstrate the use of these special components.

There might also be other significant requirements you need to consider. Your application might have to use a legacy authentication mechanism like logging on to the database. Or it might have to integrate with legacy systems for authorization or customization. Or you might need to support accessibility standards allowing your application to be used by people with disabilities. If you have these kinds of requirements, you must evaluate the risk to your project if you cannot meet them. If they are critical to your project's success, you need to validate them in a Proof of Concept.

Does the technology work?

The short answer is yes. Hundreds of organizations have already followed Oracle's lead and built big enterprise applications using Oracle ADF. It is very straightforward to use the ADF framework with relational tables—the framework handles all the boring object-relational mapping, allowing you to concentrate on building the actual application logic.

You are likely to inherit at least some of the data model from a pre-existing system, but in rare cases, you will be building a data model from scratch for a brand new application. JDeveloper does allow you to build data models, but Oracle also has other tools (for example, SQL Developer Data Modeler) that are specialized for the task of data modeling. Either way, the ADF framework does not place any specific restrictions on your data model—any good data model will work great with ADF.

But your requirements are special, of course. Nobody has ever built an application like the one you are about to build—that is the essence of a project: To do something non-trivial that has not been done before. After all, if you did not need anything special, you could just pick up a standard product off the shelf. So you need to consider all your specific requirements to see if ADF can do it.

The answer is still yes. The ADF framework is immensely powerful as it is, but it also allows you to modify the functionality of ADF applications in myriad ways to meet any conceivable requirement. If you have to work through a data access API, for instance, you can override the `doDML()` method in entity objects – allowing you to say: "Instead of issuing an UPDATE SQL statement, call this API instead." And if you need to work with existing web services for modifying data, you can create Data Sources from web services.

But you shoud not just take my word (or anybody else's word) for it. Building an enterprise application is a major undertaking for your organization, and you want to prove that your application can meet the requirements.

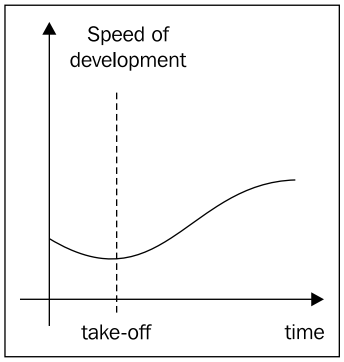
How long does it take?

It depends mainly on three things: The size of the task, the complexity of the task, and the speed of development.

The size and complexity of the task is given by your requirements. It would be a rare project where all requirements are known exactly at the beginning of the project, but if you have the set of detailed requirements, you can make a good estimate of project size and complexity.

The speed of development will be the great unknown factor if ADF is new to you and your team. Using your previous development tool (for example, Oracle Forms), you were probably able to convert your knowledge of project size and complexity into development effort—but you don't yet know what your development speed will be with ADF.

Development speed varies over time with all tools as shown next. You will often discover that your initial development speed actually decreases slightly in the beginning as you move from using the tool with all default settings to actually figuring out all the options. Then comes a learning period, and finally the take-off point where real productivity starts:



You can use your initial development speed as an approximation of the productive development speed if you need to produce a rough estimate early in the project. However, if you do this, you must be aware that there will be a period of 1-2 months of lower productivity before you start climbing up to your full productive development speed.

The Proof of Concept deliverables

The outcome of the proof of concept is not architecture in the form of boxes and arrows on a PowerPoint slide. David Clark from the Internet Engineering Task Force said, "We believe in running code" and that is what the Proof of Concept should deliver in order to be believable and credible to developers, users, and management: Running code.

If you want to convince your project review board that ADF is a viable technology, you need to bring your development laptop before your project review board and perform a live demonstration.

Additionally, it is a good idea to record the running proof of concept application with a screen-recording tool and distribute the resulting video file. This kind of demo tends to get watched in many places in the organization and gives your project visibility and momentum.

Proof of Concept case study

You are a developer with DMC Solutions—an IT company selling a system for **Destination Management Companies** (**DMC**). A DMC is a specialized travel agency, sometimes called an "incoming" agency, and works with clients in the country where it is based.

Run DMC

On an average packaged tour, you will probably not enjoy the services of a DMC. But if you manage to qualify for a company-paid trip to some exotic location, your company is likely to engage the services of a DMC at the destination. And if you have ever participated in a technology conference, a DMC will normally be taking care of transfers, dinners, receptions, and so on.

The system that DMC Solutions is selling today is based on Oracle Forms, and the sales force is saying that our competitors are offering systems with a more modern look and a more user-friendly interface. Your mission, should you choose to accept it, is as follows:

- Prove that ADF is a valid choice for a modern enterprise application
- Set up a project to build the next generation of destination management software (the **XDM** project)

The rest of this chapter shows how to build the proof of concept application, implementing two use cases. You can simply read it to get a feel for the tasks involved in creating an ADF application, or you can use it as an ADF hands on exercise and perform each step in JDeveloper on your own.

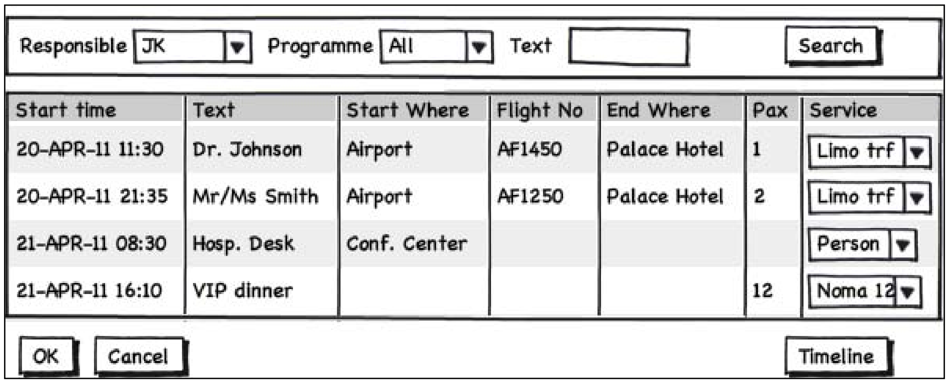
Use cases

Your manager has dusted off the specification binder for the existing system and asked you to implement Use Case 008 Task Overview and Edit. Additionally, he wants you to implement the newly specified Use Case 104 Person Task Timeline.

These two use cases represent basic application functionality (the ability to search and edit data) as well as a graphical representation of time data—something new that was not possible in Oracle Forms.

UC008 task overview and edit

This screen allows the user to search for tasks by responsible person, program or a free-text search. Data can be updated and saved back to the database:

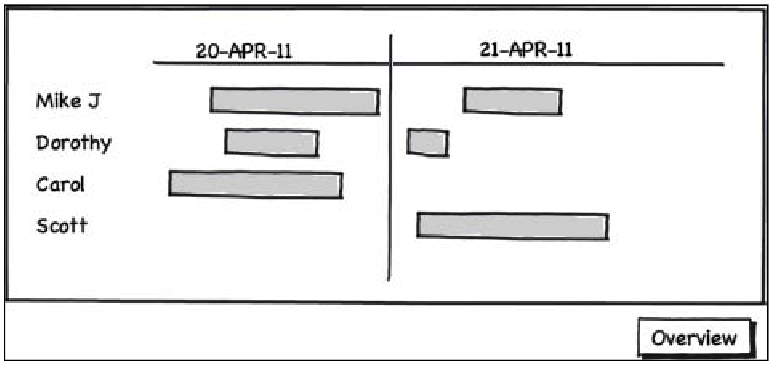


Start time	Text	Start Where	Flight No	End Where	Pax	Service
20-APR-11 11:30	Dr. Johnson	Airport	AF1450	Palace Hotel	1	Limo trf
20-APR-11 21:35	Mr/Ms Smith	Airport	AF1250	Palace Hotel	2	Limo trf
21-APR-11 08:30	Hosp. Desk	Conf. Center				Person
21-APR-11 16:10	VIP dinner				12	Noma 12

For simplicity, we will not implement the application menu, but instead just have a button labeled **Timeline** that invokes UC104.

UC104 Person Task timeline

Your manager would like something such as the Gantt charts he uses to track projects, which shows the tasks assigned to each person on a timeline:

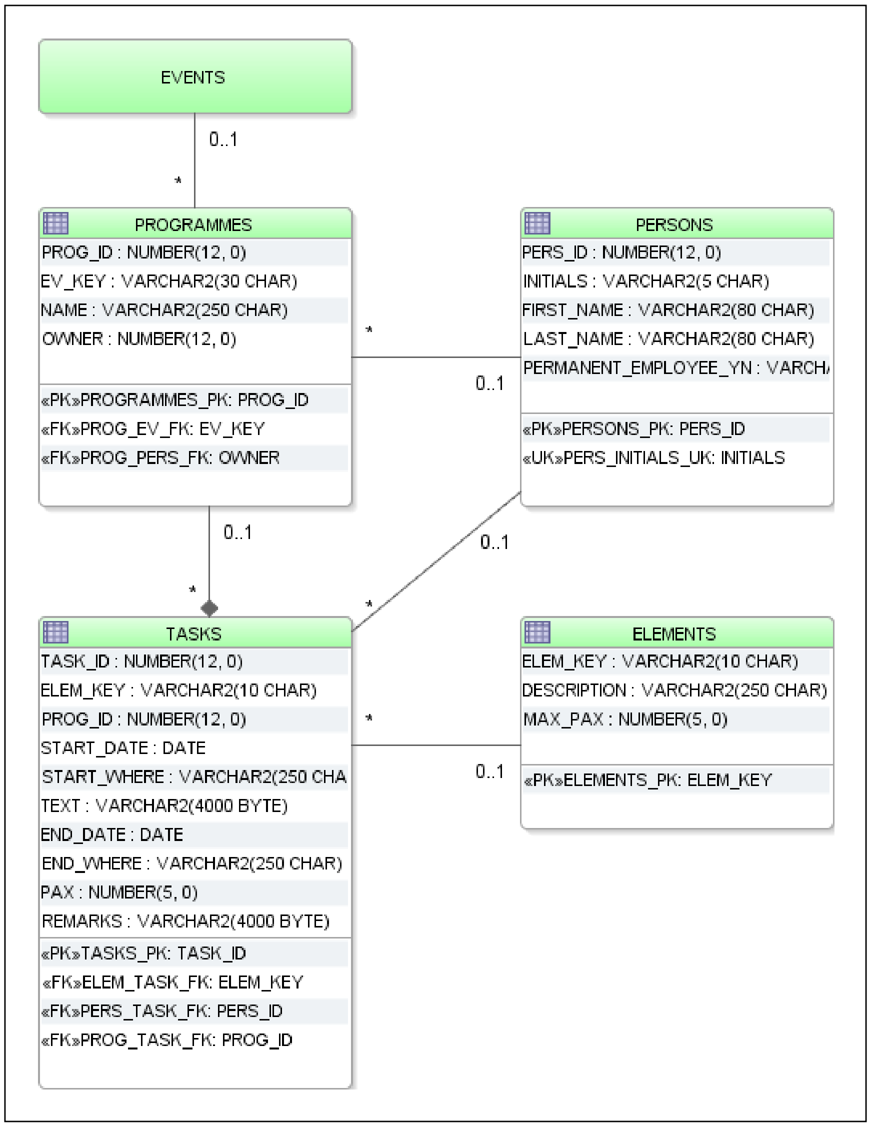


Again, we will not have a menu, just a button for returning to UC008.

Data model

The destination management system works with Events (such as "Oracle OpenWorld 2011"). Within each event, there will be a number of programs (for example, "VIP Pharma Customers"). One person is responsible for each programme. Within a programme, there will be a number of Tasks that point to standard elements from the element catalog. Examples of elements could be a Limo transfer, a dinner, an excursion, and so on. Each task will be assigned to exactly one person.

The following diagram shows just the parts of the (much larger) existing data model that we need for the Proof of Concept:


EVENTS
0..1
*
PROGRAMMES
PROG_ID : NUMBER(12, 0)
EV_KEY : VARCHAR2(30 CHAR)
NAME : VARCHAR2(250 CHAR)
OWNER : NUMBER(12, 0)
«PK»PROGRAMMES_PK: PROG_ID
«FK»PROG_EV_FK: EV_KEY
«FK»PROG_PERS_FK: OWNER
PERSONS
PERS_ID : NUMBER(12, 0)
INITIALS : VARCHAR2(5 CHAR)
FIRST_NAME : VARCHAR2(80 CHAR)
LAST_NAME : VARCHAR2(80 CHAR)
PERMANENT_EMPLOYEE_YN : VARCH
«PK»PERSONS_PK: PERS_ID
«UK»PERS_INITIALS_UK: INITIALS
TASKS
TASK_ID : NUMBER(12, 0)
ELEM_KEY : VARCHAR2(10 CHAR)
PROG_ID : NUMBER(12, 0)
START_DATE : DATE
START_WHERE : VARCHAR2(250 CHA
TEXT : VARCHAR2(4000 BYTE)
END_DATE : DATE
END_WHERE : VARCHAR2(250 CHAR)
PAX : NUMBER(5, 0)
REMARKS : VARCHAR2(4000 BYTE)
«PK»TASKS_PK: TASK_ID
«FK»ELEM_TASK_FK: ELEM_KEY
«FK»PERS_TASK_FK: PERS_ID
«FK»PROG_TASK_FK: PROG_ID
ELEMENTS
ELEM_KEY : VARCHAR2(10 CHAR)
DESCRIPTION : VARCHAR2(250 CHAR)
MAX_PAX : NUMBER(5, 0)
«PK»ELEMENTS_PK: ELEM_KEY

If you want to follow along with the proof of concept, building it in JDeveloper on your own workstation, you can download SQL scripts for creating the relevant part of the data model from the book companion website at `www.enterpriseadf.com`. This script also contains some anonymized data.

Getting started with JDeveloper

Oracle JDeveloper is Oracle's strategic development tool and the only tool with full support for ADF Development. While Oracle will continue to support NetBeans and offer a lot of functionality for Eclipse, they have also clearly stated that JDeveloper is the tool of choice for building Oracle ADF applications.

JDeveloper is freely available for download and use from the Oracle Technology Network (`otn.oracle.com`), normally through a download link from the front page. If you do not already have a free `oracle.com` account, you will have to create one.

The illustrations in this book show JDeveloper 11g Release 1, version 11.1.1.4.0, but since JDeveloper is a rapidly developing product, there might be a newer version out by the time you read the book. However, the basics have not changed over the last couple of years, and you should be able to immediately find the dialogs and options you are looking for. In addition, since Oracle has built a very large application based on this version, you can be sure that there will be a simple migration path moving forward.

The following steps describe how to create a workspace for an ADF enterprise application—if you want to use this chapter as a hands-on exercise, use the suggested values in each step:

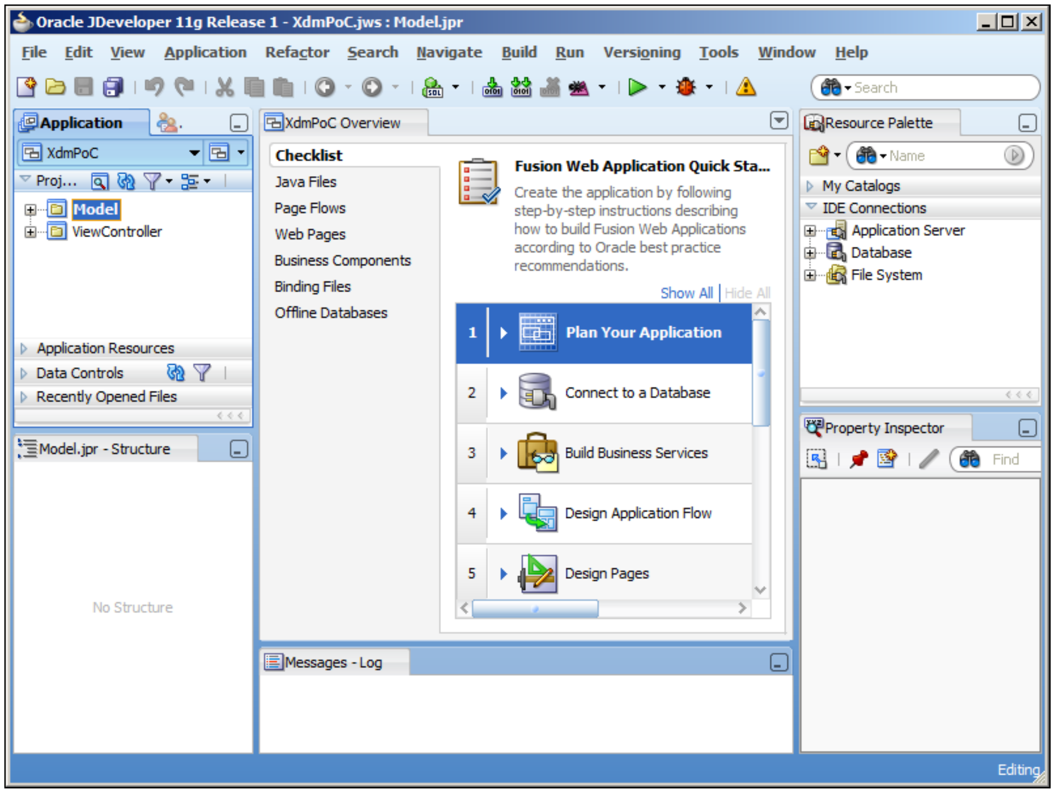
1. Start JDeveloper.
2. Choose **File | New**. In the **New Gallery** dialog box, chose **Applications** (under **General**) and choose **Fusion Web Application (ADF)**. JDeveloper offers you many other types of applications, including **Java Desktop Application (ADF)**, but you want a **Fusion Web Application**.
3. Give your application a name (for example, **XdmPoC**), choose where to put it in the file system (you can leave the default of `C:\JDeveloper\mywork\XdmPoC`), and provide an **Application Package Prefix**. Use your organization's Java prefix, followed by your project abbreviation (for the proof of concept, use `com.dmcsol.xdmpoc`).

Java Package prefix

Traditionally, package names start with your organization internet domain with the elements reversed. So, if your company domain is `mycompany.com`, your package names would all start with `com.mycompany`. However, some organizations (like Oracle) feel that their names are sufficiently unique that they don't need to include the first `com`.

If your organization has ever used Java before, your Java package prefix has probably already been chosen and documented somewhere. Ask around.

4. You can simply click **Next** through the rest of the wizard.
5. This will create you a new application containing the two projects **Model** and **ViewController**. In the main window, JDeveloper will show you the **Application Checklist** as shown next:



The application checklist actually gives a great overview of the steps involved in building an ADF application, and if you click the little triangles to expand each step, you will see links to the relevant JDeveloper functionality for that step, together with links to relevant places in the documentation. It even has checkboxes that you can check as you have completed the different phases in developing your ADF application.

The JDeveloper window and panels

JDeveloper contains a lot of different windows and panels for different purposes. The above screenshot shows the most commonly used panels, but you can toggle each of the many panels on and off using the **View** menu.

If you have not worked with JDeveloper before, please take a moment to familiarize yourself with the typical panels shown previously:

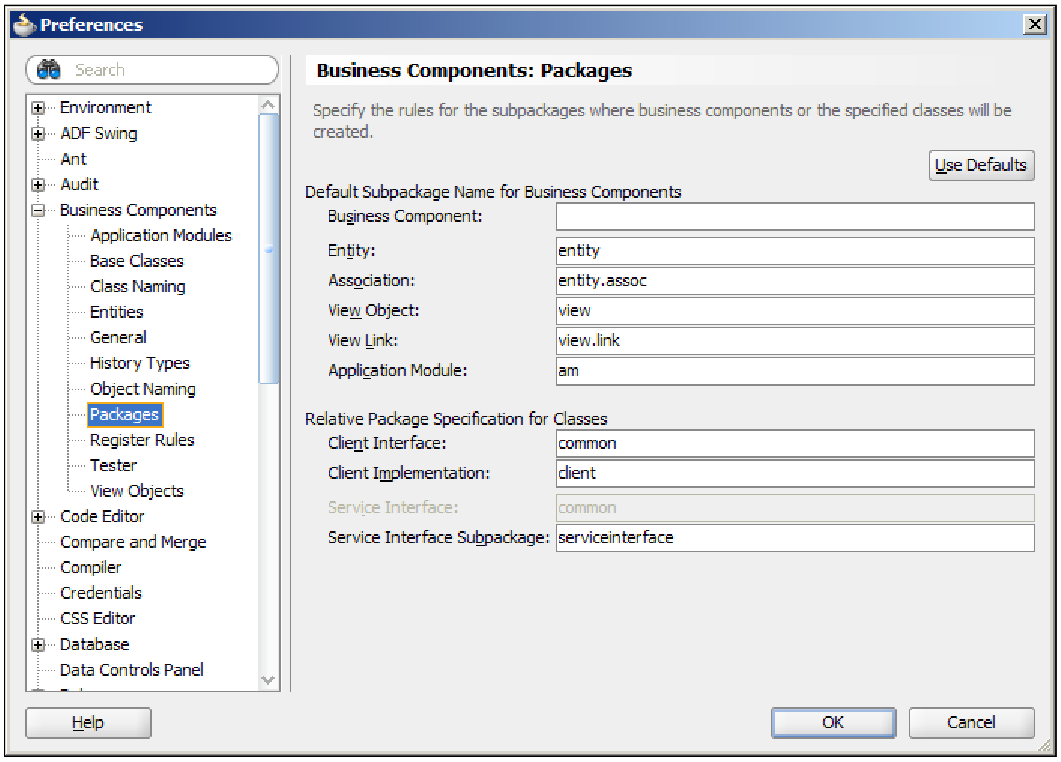
- In the middle is the main window where you will be configuring business components, designing pages, and writing code.
- Below the main window is the **Log** window showing system messages, compiling results, deployment messages and many other types of information.
- In the top left corner is the **Application Navigator**, where you can see all of the components in your workspace in a hierarchical structure.
- In the bottom left corner is the **Structure** panel. This important panel shows the detailed structure of the component you are working on. When you, for example, are working on a page in the main window, this panel will show a tree with all the components on the page.
- In the top right corner is the **Resource Palette** showing connections to application servers, databases, and so on. The **Component Palette** will also appear as a separate tab in this location when editing a page, allowing you to select components to add to the page.
- In the bottom right corner is the **Property Inspector** where you can inspect and set properties on the currently selected item.

You can rearrange these panels to your liking by grabbing the tab at the top of each panel and dragging it to a new location or even drag it out of JDeveloper to make it a floating window. This can be useful if you have multiple monitors. If you accidentally change the layout to something you do not like, you can always choose **Window | Reset Windows to Factory Settings**.

Setting JDeveloper preferences

Before you start working with JDeveloper, you should set the preferences (under **Tools | Preferences**). There are literally hundreds of references to set, most of which will not have any meaning to you yet. That's OK—the defaults are mostly fine.

One setting that you should change is the business package naming. Open the **Business Components** node and choose **Packages**. Set values for **Entity**, **Association**, **View Object**, **View Link**, and **Application Module** as shown next:



Business Components: Packages

Specify the rules for the subpackages where business components or the specified classes will be created.

Default Subpackage Name for Business Components

Business Component:	
Entity:	entity
Association:	entity.assoc
View Object:	view
View Link:	view.link
Application Module:	am

Relative Package Specification for Classes

Client Interface:	common
Client Implementation:	client
Service Interface:	common
Service Interface Subpackage:	serviceinterface

These settings tell JDeveloper to place different types of business components in separate sub packages for an easier overview when you have many components. These are just defaults to create a good starting point—as you build your application, you might decide to move your business components and classes to other packages, and JDeveloper makes this safe and easy.

You should also set **Encoding** on the **Environment** node to **UTF-8** to have all your files created in UTF-8 for maximum cross-platform portability. (If you're on Microsoft Windows, this value is probably set to a default Windows character encoding.)

Proof of Concept ADF Business Components

Once the data model is in place, the next step is to build the ADF Business Components. The description in this book is fairly brief and assumes that you have worked a little bit with ADF before, for example, by going through a basic ADF tutorial on the Oracle Technology Network website (`otn.oracle.com`). You can find links to some relevant tutorials on the book companion website `www.enterpriseadf.com`.

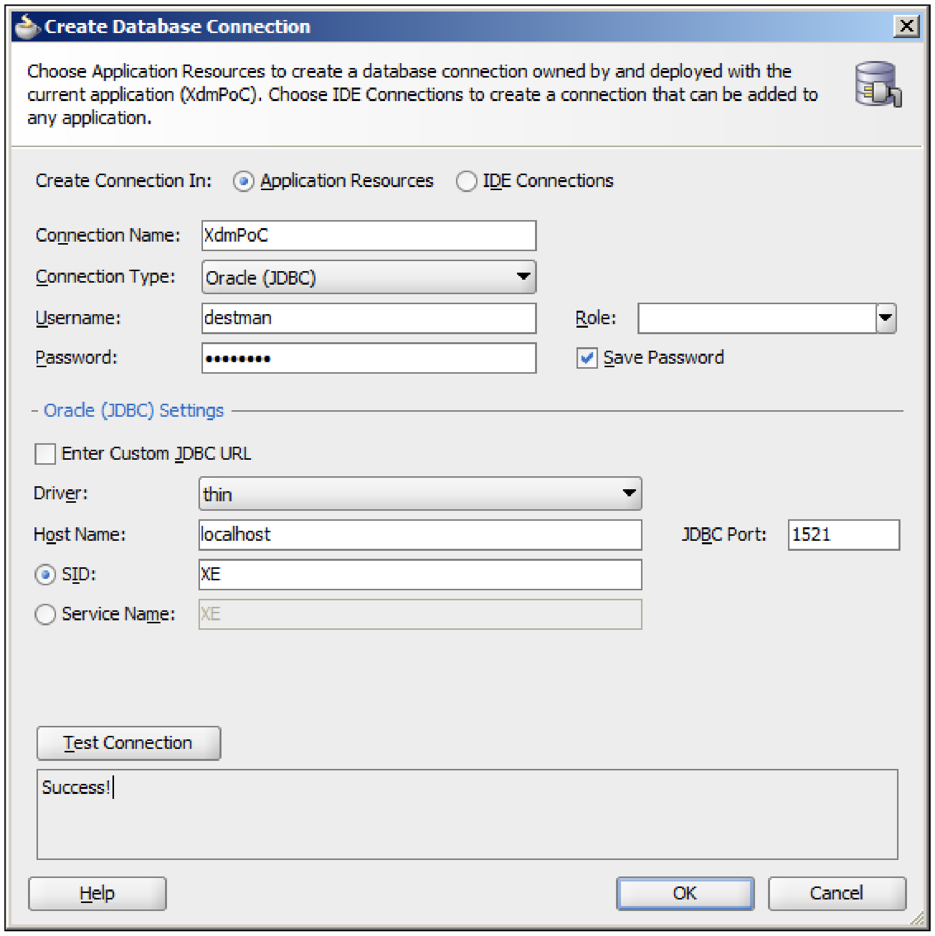
For the Proof of Concept, we will leave all business components in the default location: The **Model** project in the Proof of Concept application workspace. However, when building a real-life enterprise ADF application, you will be splitting up your application into multiple application workspaces and using ADF libraries to collect these into the master application. Working with smaller workspaces enforces modularity in the code, makes it faster for the developer to find what he's looking for, allows faster checkouts from source control – and JDeveloper also runs faster and better when not handling thousands of objects at the same time.

We will return to the proper structuring of workspaces in *Chapter 3*, *Getting Organized*.

Database Connection

As you might have noticed from the application checklist, the first step after **Plan your Application** is to create a connection to the database schema where your application tables reside. Simply choose **File | New** and in the **New Gallery** choose **Connections** (under **General**) and then **Database Connection**.

In the **Create Database Connection** dialog, give your connection a name and provide username, password, and connection information. If you are working locally with the small, free version of the Oracle Database (Oracle Express Edition), you choose the **thin** driver, enter **localhost** as **Host Name**, leave **JDBC Port** at the default value of **1521** and enter **xe** in the **SID** field. A default local installation of other database editions typically has the value **orcl** for **SID,** but is otherwise identical. If you are running against a remote database, ask your database administrator for connection information. Click **Test Connection** to check that you have entered everything correctly and then **OK**:



Choose Application Resources to create a database connection owned by and deployed with the current application (XdmPoC). Choose IDE Connections to create a connection that can be added to any application.

Create Connection In: [✓ Application Resources] IDE Connections

Connection Name:	XdmPoC		
Connection Type:	Oracle (JDBC)		
Username:	destman	Role:	
Password:	••••••••	✓ Save Password	

- Oracle (JDBC) Settings

☐ Enter Custom JDBC URL

Driver:	thin		
Host Name:	localhost	JDBC Port:	1521
● SID:	XE		
○ Service Name:	XE		

Test Connection

Success!

Building Entity Objects for the Proof of Concept

For the Proof of Concept, we will only be building entity objects for the tables that need to meet the requirements of the two use cases. To start building, select the `Model` project and choose **File | New** or right-click on the `Model` project and choose **New** from the context menu.

Make sure you select the `Model` project before you start creating business components. A default Fusion Web Application (ADF) workspace comes with two projects: A `Model` project for the business components and a `ViewController` project for the user interface.

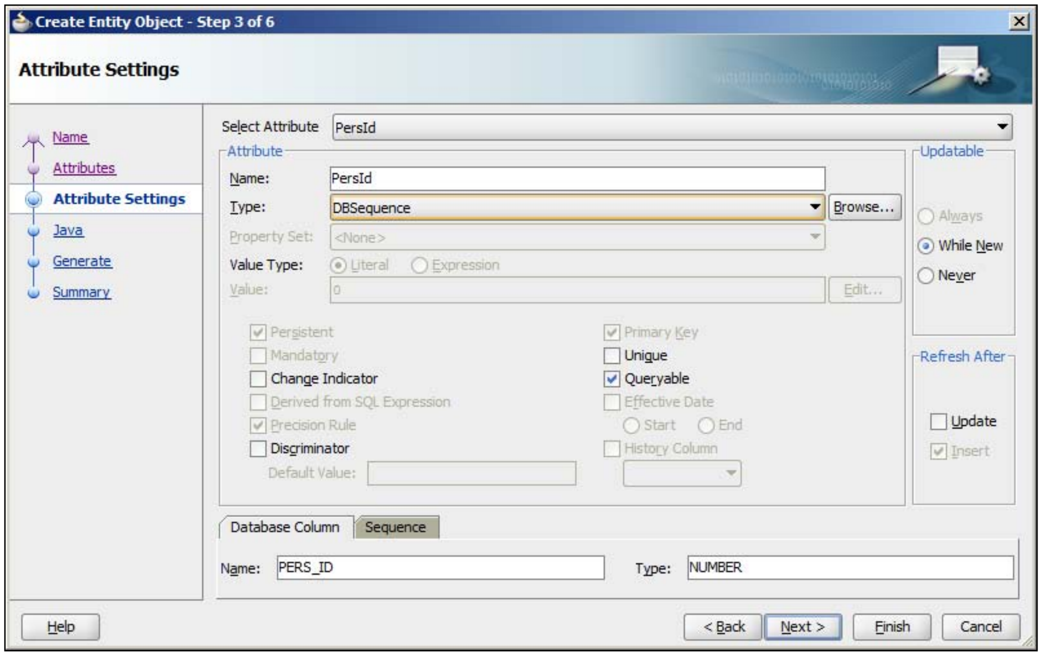
In the **New Gallery**, choose **ADF Business Components** (under **Business Tier**) and then **Entity Object**. Give the entity object a name (use the name of the corresponding table in mixed case, singular form, for example `person`) and select the corresponding schema object. You can either write the database object name in the field or click **Browse** to query the database.

Naming Standards

When you start your enterprise application development project in earnest, you need naming standards for everyone to follow. We'll return to naming standards in *Chapter 3*, *Getting Organized*.

In Step 2 of the wizard, just click **Next** to create entity object attributes for every column in the database. In ADF, there is no overhead at run time from having attributes for unused columns—when the ADF framework issues a `SELECT` statement to the database, it retrieves only the attributes that are actually needed by the view object.

In Step 3 of the wizard, you can define the entity attributes in detail. One thing that often needs to be changed here is the type for primary key columns. If the table has a numeric ID column and a database trigger that sets this value when the record is created in the database, you need to set the **Type** to **DBSequence**:



Select Attribute: PersId

Attribute
Name: PersId
Type: DBSequence
Property Set: <None>
Value Type: [✓ Literal] Expression
Value: 0

- [✓] Persistent
- [ ] Mandatory
- [ ] Change Indicator
- [ ] Derived from SQL Expression
- [✓] Precision Rule
- [ ] Discriminator
- Default Value:
- [✓] Primary Key
- [ ] Unique
- [✓] Queryable
- [ ] Effective Date
- Start End
- [ ] History Column

Updatable: Always [✓ While New] Never

Refresh After: [ ] Update [✓] Insert

Database Column | Sequence
Name: PERS_ID Type: NUMBER

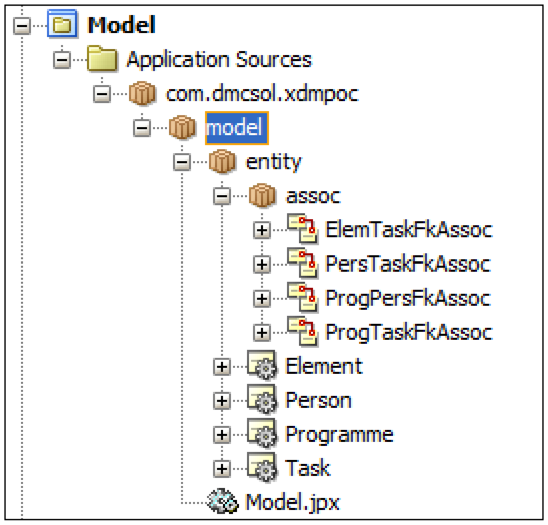
Notice that the ADF framework has now changed the values in the right-hand side of the dialog box: **Updatable** is now set to **While New**, and in the **Refresh After** box, the checkbox for **Insert** is now checked. This means that the entity object will automatically retrieve the primary key value created by your trigger. If you are using an Oracle database, ADF will use the `RETURNING` feature in Oracle SQL to get the ID back as part of the insert (without having to make a second round-trip to the database).

You do not have to make any changes in steps four through six, so you can simply click **Finish** here to close the wizard and create your entity object.

For the proof of concept, you need to follow the previous procedure to create entity objects for the following tables:

- PROGRAMMES
- TASKS
- PERSONS
- ELEMENTS (doesn't need **DBSequence** chosen)

When you are done, the **Model** project in the **Application Navigator** should look like this:



Building associations for the Proof of Concept

When you have created the entity objects, you will normally find that JDeveloper has automatically discovered the relationships between them, based on the foreign keys defined in the database.

If you have configured JDeveloper **Preferences** as recommended in the introduction, all of the associations can be found in the application navigator under `model` | `entity` | `assoc` as shown in the previous **Figure**.

The missing link
The ADF framework needs to know about the relationship between entities. In case you have to build an ADF application on an existing database where relations between data records are *not* implemented as foreign keys in the database, you can define associations in JDeveloper.

Building view objects and view links for the Proof of Concept

To determine which view objects to build, you must look at the screens you need. This allows you to determine both the data you need to present and the value lists you will need.

Looking at the Task Overview screen (UC008), we see that all data is at the same level (no master-detail), so we will just need one **Tasks** view object to display the data.

Additionally, we'll need three value lists:

- Programmes (for the **Programme** drop-down list for search)
- Persons (for the **Responsible** drop-down list for search)
- Services (for the **Service** drop-down list in the data table)

Looking at the Person Task Timeline screen (UC104), there are clearly no value lists. Because data is presented graphically, it is not immediately obvious whether the data contains any master-detail relationship. To determine if that is the case, consider how you would display the same information in ordinary fields and tables. Such a screen might show:

- One person
- A number of tasks assigned to that person

This shows us that there is actually a master-detail relationship hidden here, so we need one view object for Persons, one view object for their Tasks, and a view link connecting the two.

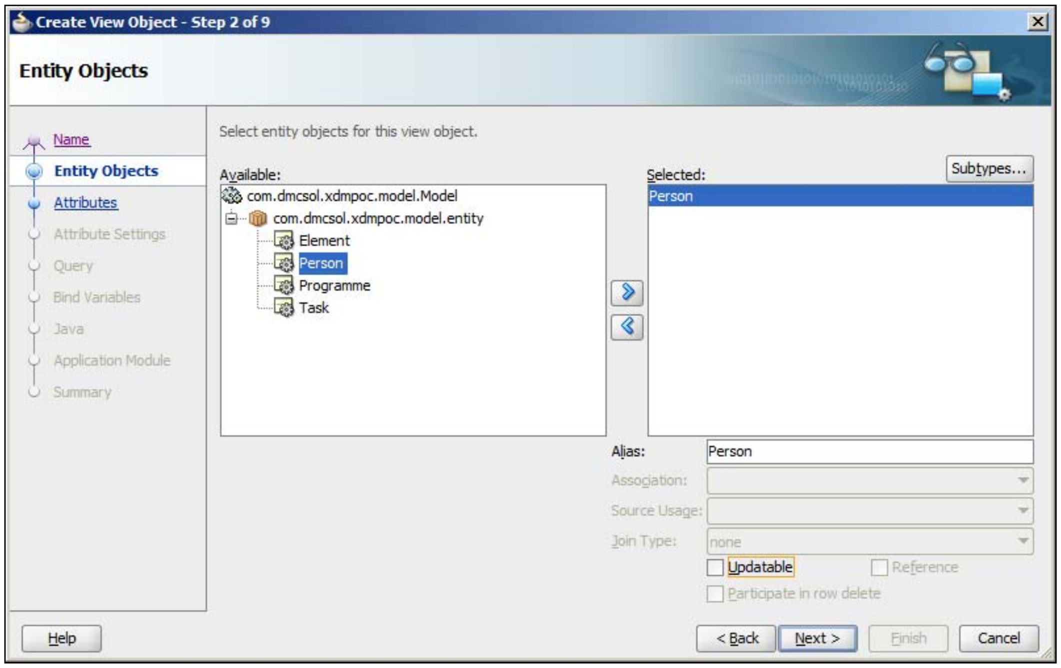
Creating view objects for value lists

To create a view objects for Persons, choose **File | New**, and in the **New Gallery** choose **View Object**. It is a good idea to give your view objects a name that in-dicates their intended usage as lists of values—for the list of persons, use the name `PersonLOV`. Leave the data source at **Updatable Access through Entity Objects**.

Always use Entity Objects

In ADF 10g and earlier, the recommendation was to use **Read-Only Access through SQL Query** when you did not need to change the data. In ADF 11g, the benefit from caching that entity objects offer outweighs the slight performance benefit from executing SQL directly. The recom-mendation is, therefore, to always access data through entity objects.

In step 2 of the wizard, choose the `Person` entity object and move it to the box on the right. You can remove the checkmark in the **Updatable** box, since we will only be using this view object for the drop-down list:



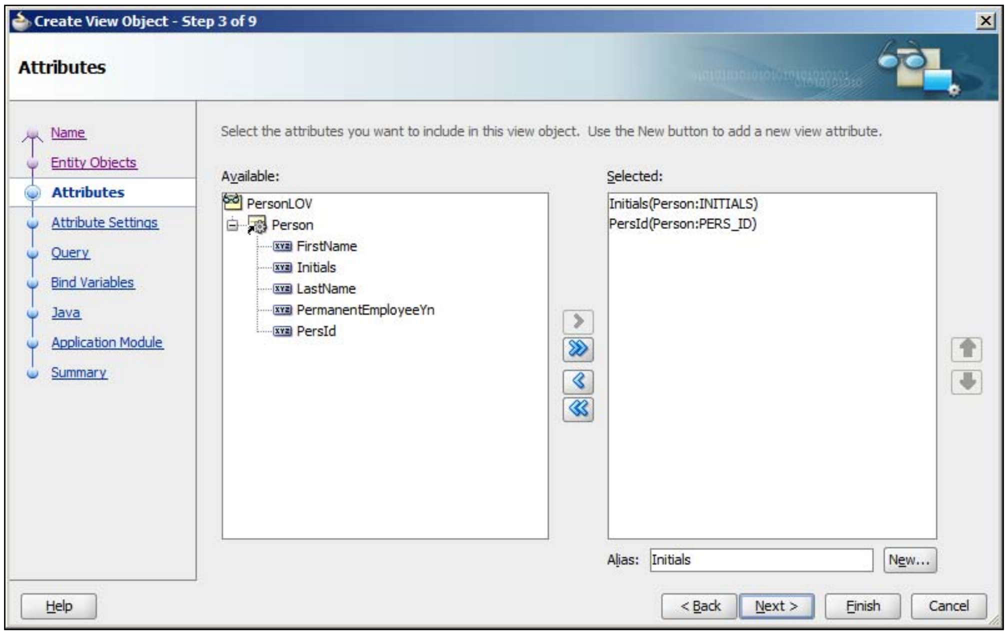
Select entity objects for this view object.

Available:
- com.dmcsol.xdmpoc.model.Model
  - com.dmcsol.xdmpoc.model.entity
    - Element
    - Person
    - Programme
    - Task

Selected:
- Person

Alias: Person
Association:
Source Usage:
Join Type: none
[ ] Updatable [ ] Reference
[ ] Participate in row delete

In step 3 of the wizard, move the fields you want to the right-hand side—note that the primary key attribute will always be included:



Select the attributes you want to include in this view object. Use the New button to add a new view attribute.

In step 5 (**Query**), you define the ordering of records by entering **initials** in the **Order By** field. Then click **Finish** to create the view object.

Repeat this procedure to create the two other value list view objects:

- `ProgrammeLOV` (based on the `Programme` entity object, select the attribute called `name`, order by `name`)
- `ServiceLOV` (based on the `Element` entity object, select the attribute `description`, order by `description`)

Creating a view object for tasks

To create a view object for tasks, look at the Task Overview and Edit page (UC008) and at the data model. You will notice that we need fields for date and time, text, start where, flight number, end where, number of passengers, and service. All of this data comes from the `TASKS` table through the `Task` entity object.

Create a new view object (`AllTasksVO`), leaving the data source at **Updatable Access through Entity Objects**. In step 2 of the wizard, choose the `Task` entity object and move it to the right-hand side. Because we will actually be updating data through the `AllTasksVO` view object, we leave the check mark in the **Updatable** checkbox.

In Step 3, shuttle the following fields to the right hand side:

- **StartDate**
- **Text**
- **StartWhere**
- **FlightNo**
- **EndWhere**
- **Pax**
- **ElemKey**

Note that in the **Available** box on the left-hand side, all attributes are shown in al-phabetical order, not in the order you placed them in the entity objects or the order they have in the database table.

Click **Next** two times and choose to order by `start_date`. Then click **Next** to get to **Bind Variables** (Step 6).

Bind variables

Bind variables are placeholders in your SQL that you fill with values at run time. The ADF framework enforces the good practice to always use bind variables when you need to change the `WHERE` condition of a query. You should never simply concatenate values into an SQL statement; if you do, the database cannot tell that it already knows the SQL statement and will waste time parsing it again, and a malicious user could potentially insert extra statements into your SQL.

Looking at the search box at the top of the screen sketch, you can see that we need to limit the tasks displayed by responsible person, programme, and text. Use the **New** button to create three bind variables called `pResponsible`, `pProgramme` and `pText`. You can leave the other settings in this step of the wizard at their default values.

When you're done, click the **Back** button to return to step 5 of the wizard, and add a `WHERE` clause that uses the bind variables. It should look like this:

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

```
(:pResponsible is null or PERS_ID = :pResponsible)
and (:pProgramme is null or PROG_ID = :pProgramme)
and (:pText is null or upper(TEXT) like '%' || upper(:pText) || '%')
```

When you are done entering the WHERE clause, click on the **Test** button to verify that your SQL is valid.

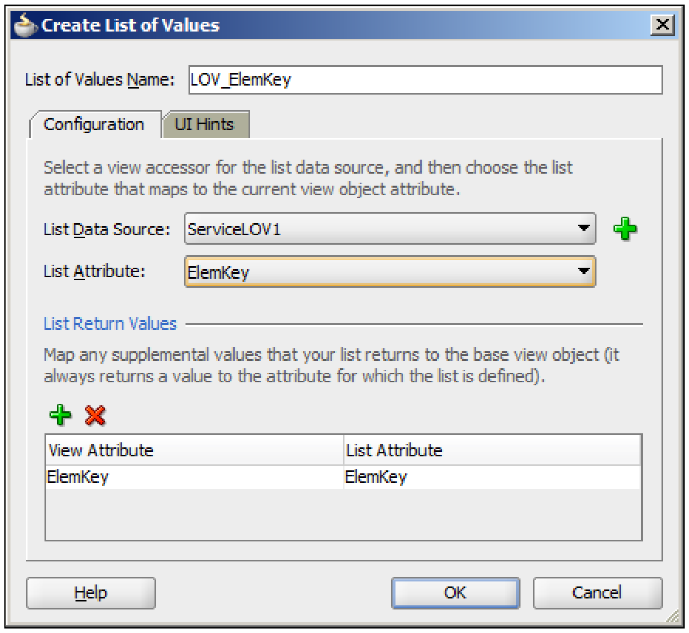
In this case, we allow null values for the bind variables, so the SQL statement has to contain an OR branch handling this case. We are converting both the database TEXT column and the pText bind variable to upper case to achieve case-insensitive matching, and concatenating a wildcard character before and after the parameter value to search for occurrences of the search text anywhere in the database value.

Point-and-click Where clauses

You can also define **Named View Criteria** on your view objects (on the **Query** sub-tab). These allow you to build a where clause by pointing and clicking. Read about named view criteria and the associated af:query component in the online help.

When you click **Finish**, the AllTasksVO view object is created and appears in the application navigator.

However, we are not quite done with the view object—we still need to define which data elements use lists of values. You might remember from the page layout illustration that **Service** was rendered as a drop-down list box. Double-click on the AllTasksVO view object to edit it and choose the **Attributes** sub-tab. Choose the ElemKey attribute and then click on the green plus sign next to the heading **List of Values**. The **Create List of Values** dialog appears:



List of Values Name: LOV_ElemKey

Select a view accessor for the list data source, and then choose the list attribute that maps to the current view object attribute.

List Data Source: ServiceLOV1

List Attribute: ElemKey

List Return Values

Map any supplemental values that your list returns to the base view object (it always returns a value to the attribute for which the list is defined).

View Attribute	List Attribute
ElemKey	ElemKey

In this dialog, click the green plus sign to add the **List Data Source**. Choose the `ServiceLOV` view object and then `ElemKey` as **List Attribute**.

Since we do not want to display the actual key value (`ElemKey`) to the user, you should click on **UI Hints** tab, and move the `Description` attribute to the right-hand box. Uncheck the **Include No Selection** checkbox, and then click **OK**.

The **Attributes** tab also allows you to define some hints to the user interface components about rendering the component. Double-click the `StartDate` attribute and choose the **Control Hints** sub-tab. Set the Label Text to `Start time`, set **Format Type** to **Simple Date** and in the **Format** field, enter the format mask `dd-MMM-yy HH:mm`.

The date format string used here is Java `SimpleDateFormat`—not the SQL data format strings you might be used to from the database.

Click **OK** and then set a **Label Text** for the remaining elements, referring to the user interface sketch for UC008 (**Format** is only used for date and number objects).

Taking a hint

The **Control Hints** defined here are only that: Hints. When building the user interface, these will be default, but you can still decide to use another label text or format.

Building an application module for tasks

To create an application module for tasks, choose **File | New** and then **Application Module**. Give the application module a name (for example, `EditTaskService`). In step 2 of the wizard, expand the tree in the left-hand side and shuttle the `AllTasksVO` to the right-hand side, together with the `PersonLOV` and `ProgrammeLOV` that we need to create the search criteria value lists. This is all you need to do, so you can simply click **Finish** to close the wizard.

Now you can verify that your application module works the way you expect it to. In the application navigator, right-click on the `EditTaskService` application module node (the icon that looks like a little suitcase), and choose **Run** from the context menu. This will start the Business Components Tester where you can work with all of the view objects that are part of your application module.

Always test your business components using the Business Components Tester before you start using them in pages. Whenever your page does not run the way you expect, always use the Business Components Tester to determine if the error is in the frontend or the backend part of the application.

Double-click on the `AllTasksVO1` view object. A pop-up dialog appears, allowing you to assign values to all the bind variables defined in the view objects. To begin, just click **OK** to leave all bind variables at the value `NULL`.

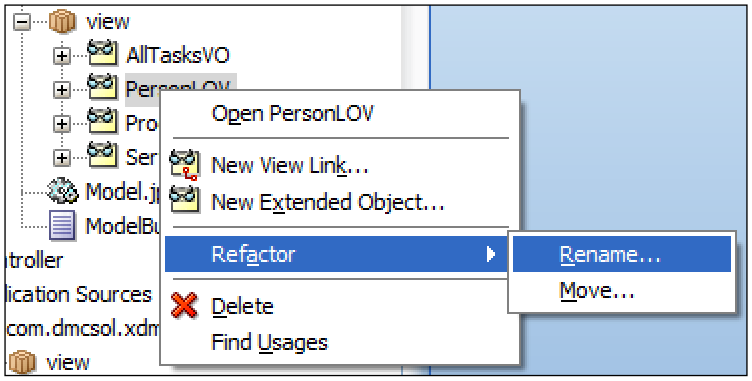
Here, you can page through the existing data as well as insert and delete rows using the green plus and the red cross symbol. Click on the **Edit Bind Variables** button (to the right of the toolbar, with the little pencil icon) to change bind variable values and notice how data is filtered.

Creating view objects for scheduling

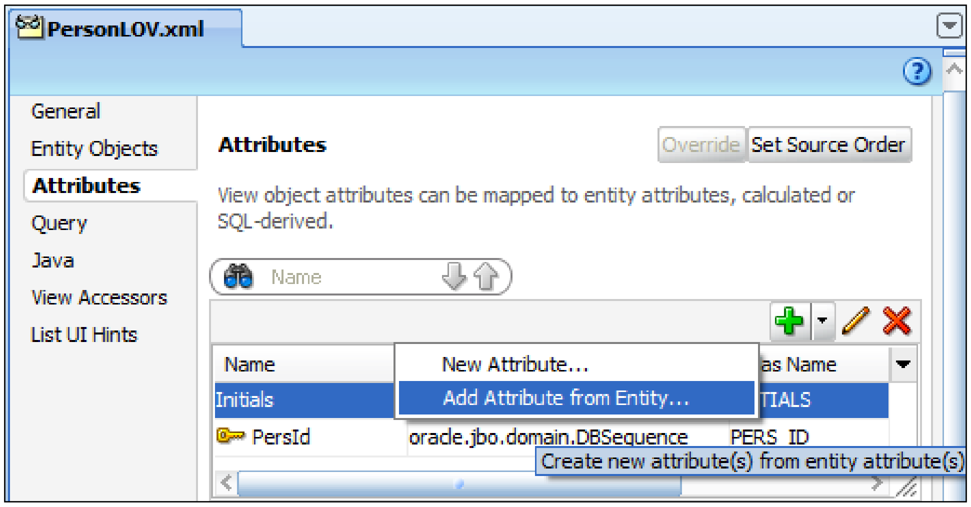
For the scheduling screen, we need two view objects: one for persons, and one for tasks assigned to persons.

We already have a view object showing persons, but this view object only contains initials (because it was intended for the Persons drop-down in UC008). We could create a new persons view object for UC104, but we will change the existing view object instead.

First, you need to change the name of the view object from `PersonLOV` to `PersonsVO` to reflect that it is no longer just used for a list of values. Changing name or package for existing objects is called **refactoring**, and JDeveloper makes this easy. Simply right-click on the `PersonLOV` view object and choose **Refactor | Rename** from the context menu, JDeveloper will change the name of the object and automatically update all references to it:



Next, the view object needs some more attributes. To add these, open the view object by double-clicking on it and choose the **Attributes** sub-tab. Click the little triangle next to the green plus sign above the attributes and choose **Add Attribute from Entity**. Do not just click the plus sign—you need to select the little triangle to get access to the **Add Attribute from Entity** menu item you need:



View object attributes can be mapped to entity attributes, calculated or SQL-derived.

Name		as Name
Initials		TIALS
PersId	oracle.jbo.domain.DBSequence	PERS_ID

In the **Attributes** dialog, add the `FirstName` and `LastName` attributes to the **Selected** list. Then select the new `FirstName` and click the little pencil icon to edit the attributes. Under **Control Hints,** set a **Label Text**. Repeat for `LastName`**.**

Then create another view object, giving it the name `ScheduledTasksVO`. In step 2 of the wizard, move the `Task` entity object to the right-hand side. Since we will not be updating tasks either, you can remove the checkbox in the **Updatable** field here as well. In step 3 of the wizard, you only need to select the `StartDate` and `EndDate` attributes – note that the `TaskId` primary key attribute is automatically added.

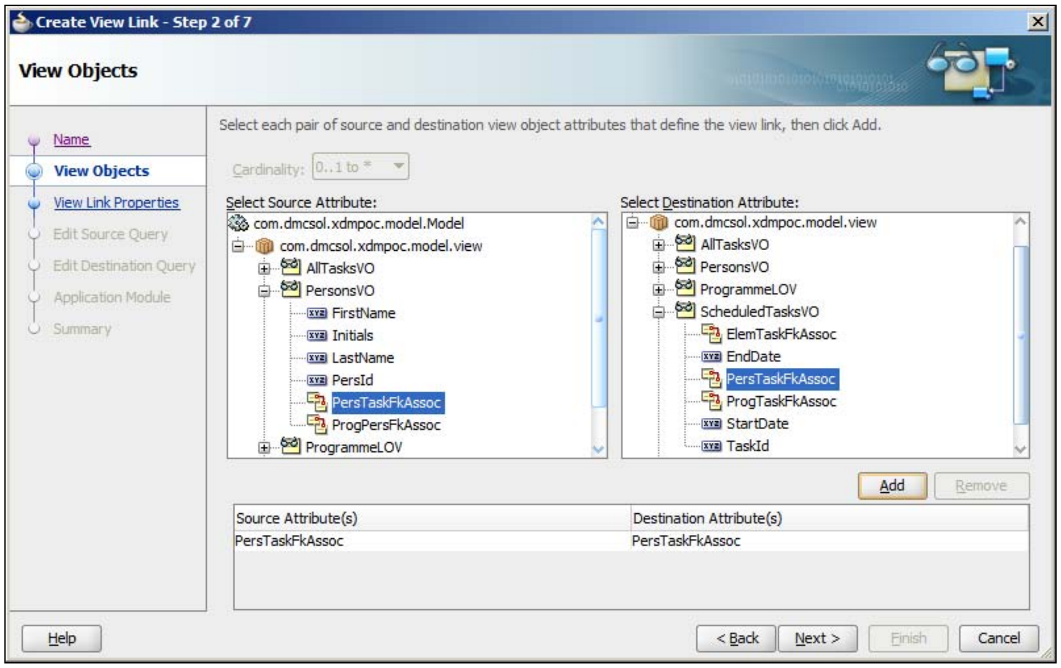
In step 5 of the wizard, we need to add a `WHERE` clause so that the view object will only show tasks with both a start and an end date. Enter the following `WHERE` clause:

```
start_date is not null and end_date is not null
```

Then click **Finish**.

Since there is a master-detail relationship between persons and tasks, we also need to create a view link. Select **File | New** and then **View Link**. Give your view link the name `PersonsTasksLink`.

In step 2 of the wizard, we need to define the relationship between the two view objects. These are connected by the foreign key `PERS_TASK_FK` that defines the connection between a person and the tasks assigned to that person. Expand the `PersonsVO` node on the left and choose `PersTaskFkAssoc` as the left-hand side of the link. On the right, expand the `ScheduledTasksVO` node and again choose `PersTaskFkAssoc`, this time as the right-hand side of the link. Then click **Add**. You see the source and destination attributes added at the bottom of the dialog box:



Select each pair of source and destination view object attributes that define the view link, then click Add.

Cardinality: 0..1 to *

Select Source Attribute:

- com.dmcsol.xdmpoc.model.Model
- com.dmcsol.xdmpoc.model.view
  - AllTasksVO
  - PersonsVO
    - FirstName
    - Initials
    - LastName
    - PersId
    - PersTaskFkAssoc
    - ProgPersFkAssoc
  - ProgrammeLOV

Select Destination Attribute:

- com.dmcsol.xdmpoc.model.view
  - AllTasksVO
  - PersonsVO
  - ProgrammeLOV
  - ScheduledTasksVO
    - ElemTaskFkAssoc
    - EndDate
    - PersTaskFkAssoc
    - ProgTaskFkAssoc
    - StartDate
    - TaskId

Source Attribute(s)	Destination Attribute(s)
PersTaskFkAssoc	PersTaskFkAssoc

You do not need to change any of the remaining settings in this wizard, so you can simply click **Next** and **Finish** to close the dialog box.

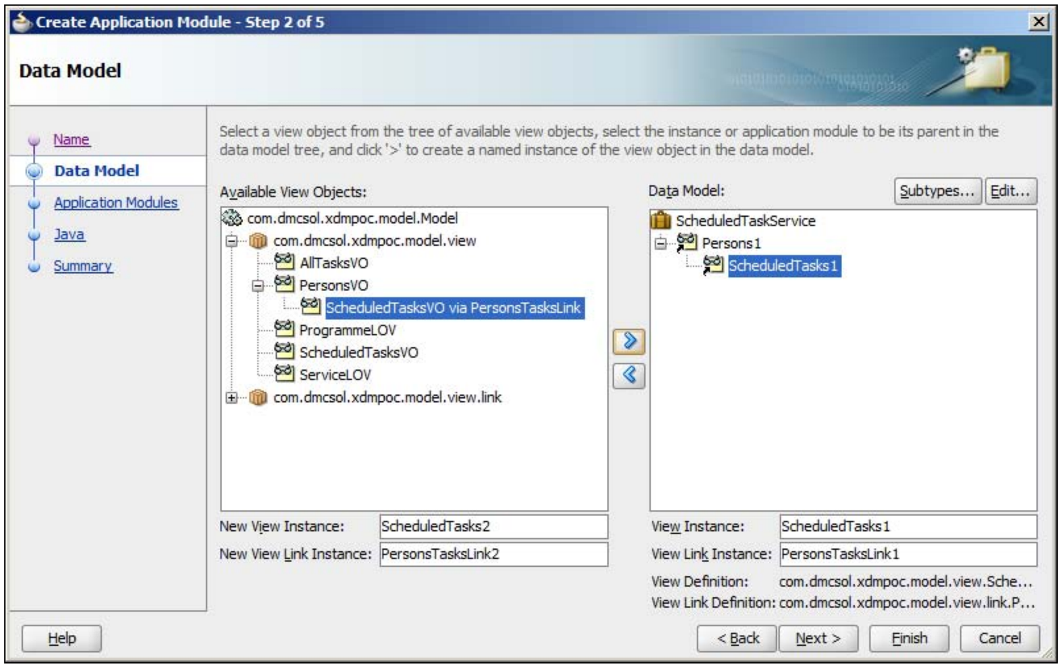
Building an application module for scheduling

Create another application module for the UC104 Person Task Timeline screen, giving it the name `ScheduledTaskService`.

One or two lumps?

On one hand, each application module uses a database connection, and your database administrator will tell you to keep this number down. On the other hand, you want to modularize your application so that each piece of functionality is completely developed and delivered by one team—this calls for multiple application modules. We'll return to the discussion of the proper number of application modules in *Chapter 3*, *Getting Organized*.

In step 2 of the wizard, first move the `PersonsVO` to the right-hand side. Then select the `Persons1` view object instance on the right and the node **ScheduledTasksVO** via **PersonTaskLink** on the left, and click the **>** button to move `ScheduledTasks` to the right-hand box:



Select a view object from the tree of available view objects, select the instance or application module to be its parent in the data model tree, and click '>' to create a named instance of the view object in the data model.

Available View Objects:

- com.dmcsol.xdmpoc.model.Model
  - com.dmcsol.xdmpoc.model.view
    - AllTasksVO
    - PersonsVO
      - ScheduledTasksVO via PersonsTasksLink
    - ProgrammeLOV
    - ScheduledTasksVO
    - ServiceLOV
  - com.dmcsol.xdmpoc.model.view.link

Data Model:

- ScheduledTaskService
  - Persons1
    - ScheduledTasks1

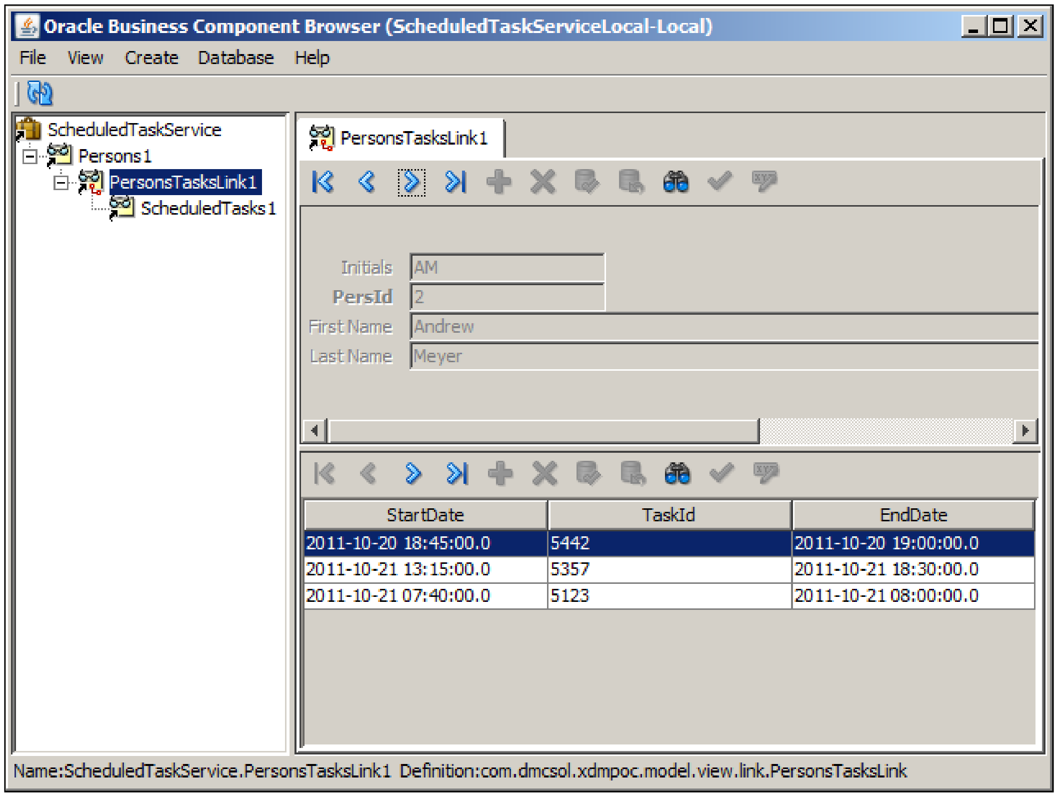
New View Instance: ScheduledTasks2
New View Link Instance: PersonsTasksLink2

View Instance: ScheduledTasks1
View Link Instance: PersonsTasksLink1
View Definition: com.dmcsol.xdmpoc.model.view.Sche...
View Link Definition: com.dmcsol.xdmpoc.model.view.link.P...

Note the difference between choosing `ScheduledTasksVO` on its own and choosing `ScheduledTasksVO` as a child of `PersonsVO`. If you choose the view object as a child of another view object, the ADF framework will automatically implement the master detail relationship – the view object will only display the records that are children of the current record in the parent (master) view object. If you choose the view object on its own, it will not have any relationship to the master view object and will simply display all child records.

Then click **Finish** to close the wizard.

Run your new application module in the business components tester. In the left-hand side, you will see the master view object, the view link, and the detail view object. Double-click on the view link to see master and detail records together. When you use the navigation buttons at the master level, you will see different detail records displayed:



Initials	AM
PersId	2
First Name	Andrew
Last Name	Meyer

StartDate	TaskId	EndDate
2011-10-20 18:45:00.0	5442	2011-10-20 19:00:00.0
2011-10-21 13:15:00.0	5357	2011-10-21 18:30:00.0
2011-10-21 07:40:00.0	5123	2011-10-21 08:00:00.0

Name:ScheduledTaskService.PersonsTasksLink1 Definition:com.dmcsol.xdmpoc.model.view.link.PersonsTasksLink

Proof of Concept ADF user interface

Once you have built all the business components your application will need, you can start building the user interface. The user interface consists of two parts: **ADF Task Flows** and **ADF Pages**.

Pages or fragments?

As mentioned in the section on ADF architecture, an application can use either pages or page fragments. The Proof of Concept uses pages for simplicity, while the professional Proof of Concept we will be building in *Chapter 6*, *Building the Enterprise Application* will use page fragments.

Creating ADF task flows

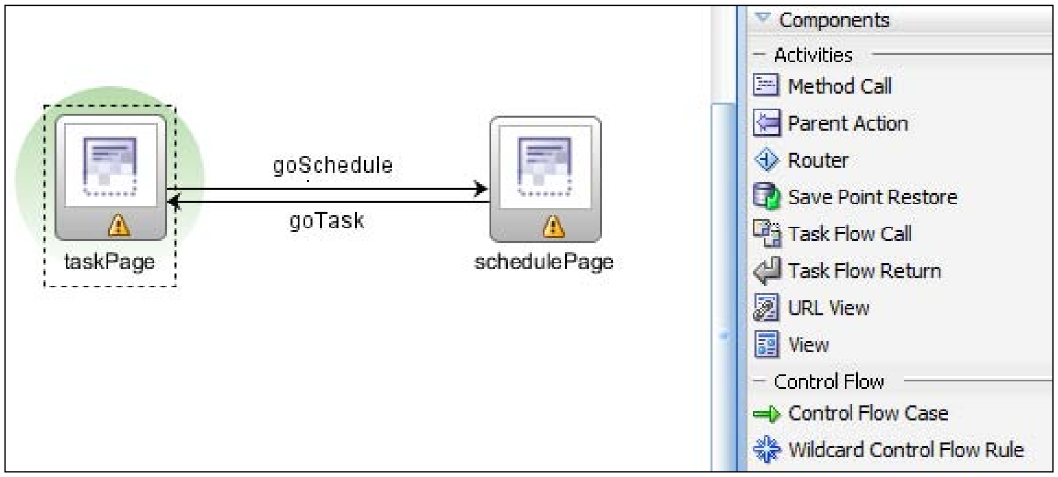
For the proof of concept, we will implement one bounded ADF task flow. Select the `ViewController` projects and then choose **File | New**. You might notice that the **New Gallery** looks differently now. That is because the `ViewController` project is active, and this project uses different technologies.

Under **Web Tier**, choose **JSF** and then **ADF Task Flow**. Give your task flow a name (for example, `xdm-poc-flow`), make sure the **Create a Bounded Task Flow** checkbox is checked and the **Create with Page Fragments** checkbox is *not* checked. Then click **OK**. You will see a blank task flow diagram in the JDeveloper main window.

In the component palette in the top right corner of the JDeveloper window, expand the **Components** heading and drag in a **View** component. Give it the name `taskPage`. Drag in another **View** components and give it the name `schedulePage`.

Then drag in a **Control Flow Case** components and drop it on the `taskPage`. Move the cursor to the `schedulePage` and click. This establishes a control flow from the `taskPage` to the `schedulePage`. The cursor will be placed in a box in the middle of the line. Type `goSchedule` in this box and press *Return*. Drop another **Control Flow Case** onto the `schedulePage` and drag it to the `taskPage`. Give this control flow the name `goTask`.

This defines the two pages that we will be using in the proof of concept, as well as the possible navigation between them. Your task flow should look like this:



Note the green halo behind the `taskPage`. That indicates that this is the default activity—the first screen presented to the user when the task flow is run. You can set another page as the default activity by right-clicking on it and choosing **Mark Activity** and then **Default Activity**.

The tasks page

You will notice that both the pages in the task flow diagram have a little yellow exclamation mark. That indicates that the pages do not actually exist yet, they are only defined as placeholders in the task flow.

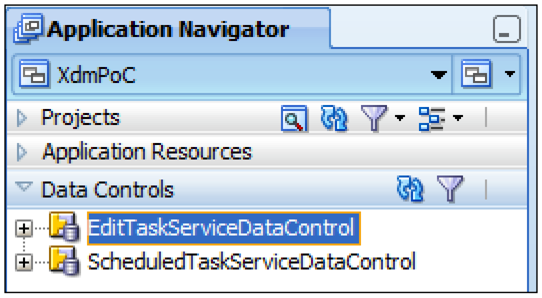
Creating the tasks page

To create the tasks page, double-click on the `taskPage` icon in the page flow diagram. The **Create JSF Page** dialog appears. For the proof of concept, we start from blank pages (make sure **Blank Page** is selected) – but when actually building the real application, we will, of course, be using page templates. Click **OK** to create and open the page.

There are two ways to place ADF components on a JSF page: you can drag them in from the component palette on the right, or you can drag them in from the data binding palette on the left.

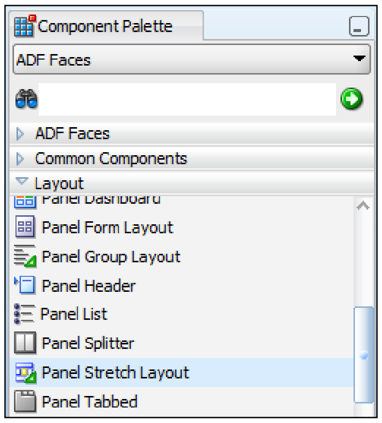
If you drag in a component from the component palette, it is not bound to any data control. This means it has no connection to the data in the ADF business components.If you drag in the data control from the data control palette, JDeveloper will automatically present you with a menu of components that you can drop onto the page. If you use this approach, the dropped component is automatically bound to the data control you dragged in.

To add components to the task page, find and open the **Data Controls** panel in the left-hand side of the JDeveloper window. You should see two data controls, corresponding to the two application modules you have created in the **Model** project: `ScheduledTaskServiceDataControl` and `EditTaskServiceDataControl`.



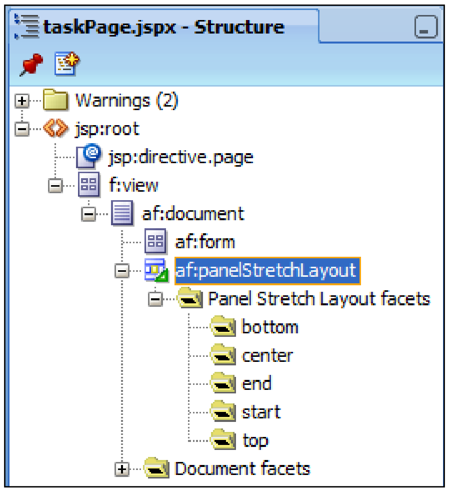
However, before we start dragging in data controls, we need to place a layout component on the page to control where items are to be placed. If you come from a 4GL background (like Oracle Forms), you might be used to pixel-precise placement of items. In JSF, on the other hand, the placement of components is controlled by special layout components. This has the advantage that the layout components can arrange, shrink, and expand the components they contain in order to make the best use of the available screen area. The disadvantage to this approach is that it takes a little while to learn to use the right layout components.

For the `taskPage`, we start with a **Panel Stretch Layout**. Find this component in the **Component Palette** in the right-hand side of the JDeveloper window (under the **Layout** heading) and drop it on the page:



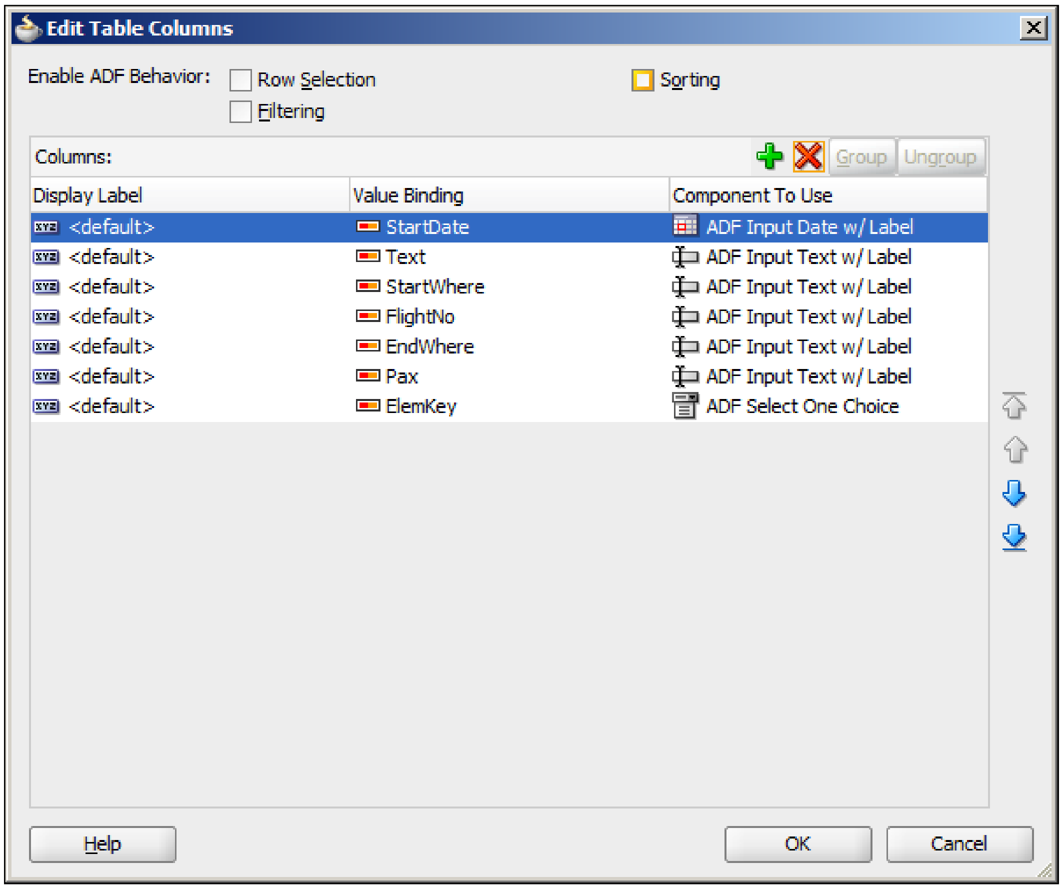
It is a good idea to use a "stretchable" layout component as the outer layer to ensure that your application will utilize the entire browser window.

If you expand the **Panel Stretch Layout** in the **Structure Panel** at the bottom right of the screen, you will see that it shows a folder-like node called **Panel Stretch Layout Facets**, and under that additional folder-like nodes called **bottom**, **center**, and so on. Many layout containers contain these containers (called **facets**) that you can place your content in:



If you refer back to the screen design earlier in this chapter, you see that the **Panel Stretch Layout** matches our requirements: We can place the search criteria on top (in the facet called **top**), the actual data in the middle (in the facet called **center**), and some buttons at the bottom (in the facet called **bottom**). We do not need the **start** and **end** facets, but don't have to worry about them—facets without content are not shown at run time.

We will start with the actual data, which we will present using an ADF Table component. Open the **Data Controls** panel on the right, and then open the `EditTaskServiceDataControl` node. You see the `AllTasksVO1` view object. Grab the entire view object and drag it onto the center facet. When you drop a data control onto a page, JDeveloper shows a context menu asking you which user interface elements you want to create and bind to the data control. In this case, select first **Table** and then **ADF Table** from the context menu. The **Edit Table Columns** dialog appears:



Display Label	Value Binding	Component To Use
<default>	StartDate	ADF Input Date w/ Label
<default>	Text	ADF Input Text w/ Label
<default>	StartWhere	ADF Input Text w/ Label
<default>	FlightNo	ADF Input Text w/ Label
<default>	EndWhere	ADF Input Text w/ Label
<default>	Pax	ADF Input Text w/ Label
<default>	ElemKey	ADF Select One Choice

In this dialog box, you can remove the columns that you do not need, and re-order the columns if necessary. You will see that JDeveloper has automatically selected an appropriate UI component—ADF Input Date for date attributes or ADF Select One Choice for attributes where a value list has been defined. For the tasks table, you only need to delete the `TaskId` column and click **OK**. You will see a table component in the middle of your page.

Finally, you need to tell ADF which column gets to use any extra space on the screen. Remember that we started with a Panel Stretch Layout, which will automatically stretch the components it contains – but a table component does not stretch until you specify which column should expand to use any extra space.

First select the **Text** column and make a note of its **Id** property (look in the **Property Inspector** in the lower right corner of the JDeveloper window) – it will be something like `c3`. Then select the entire table (either in the Design window in the middle of JDeveloper or in the **Structure Panel** at the lower left). The **Property Inspector** will now show the properties of the table. Expand the **Appearance** node and set the **ColumnStretching** property to the name of the `Text` column (for example, **column:c3**).

Running the Initial Tasks Page

Even though we do not have the search functionality built yet, it is about time that we run some code. Simply right-click anywhere on the `taskPage` page and choose **Run** from the pop-up menu. In the log window at the bottom of the JDeveloper window, you can see the WebLogic server starting up. This will take a while.

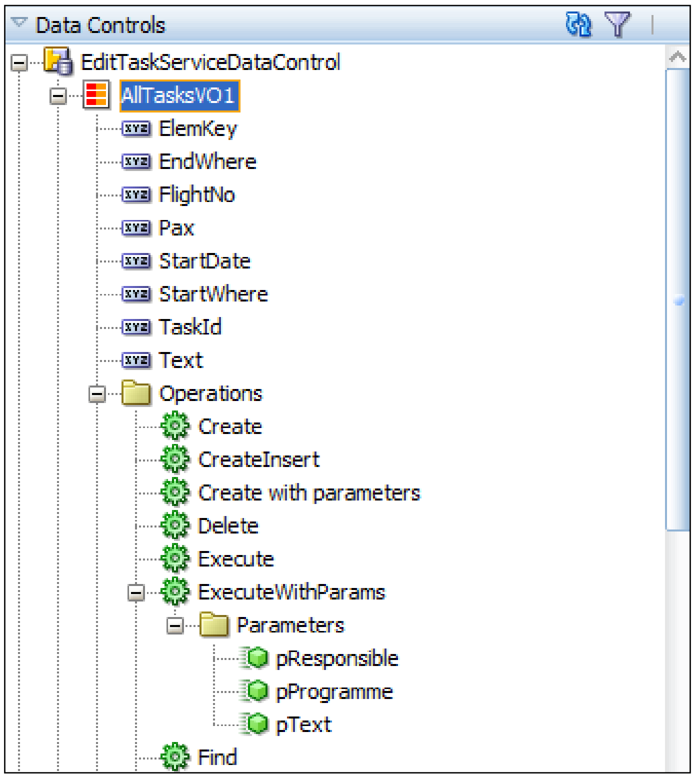
Once WebLogic has started, a browser window will open, showing your data. Resize the window, checking that the Text column expands and contracts.

You might want to change the initial column size for some of the columns—to do this, in JDeveloper select an **af:column** element in the main window or the structure panel and change the **Width** value (under the **Appearance** heading) in the **Property Inspector**.

Refining the Tasks Page

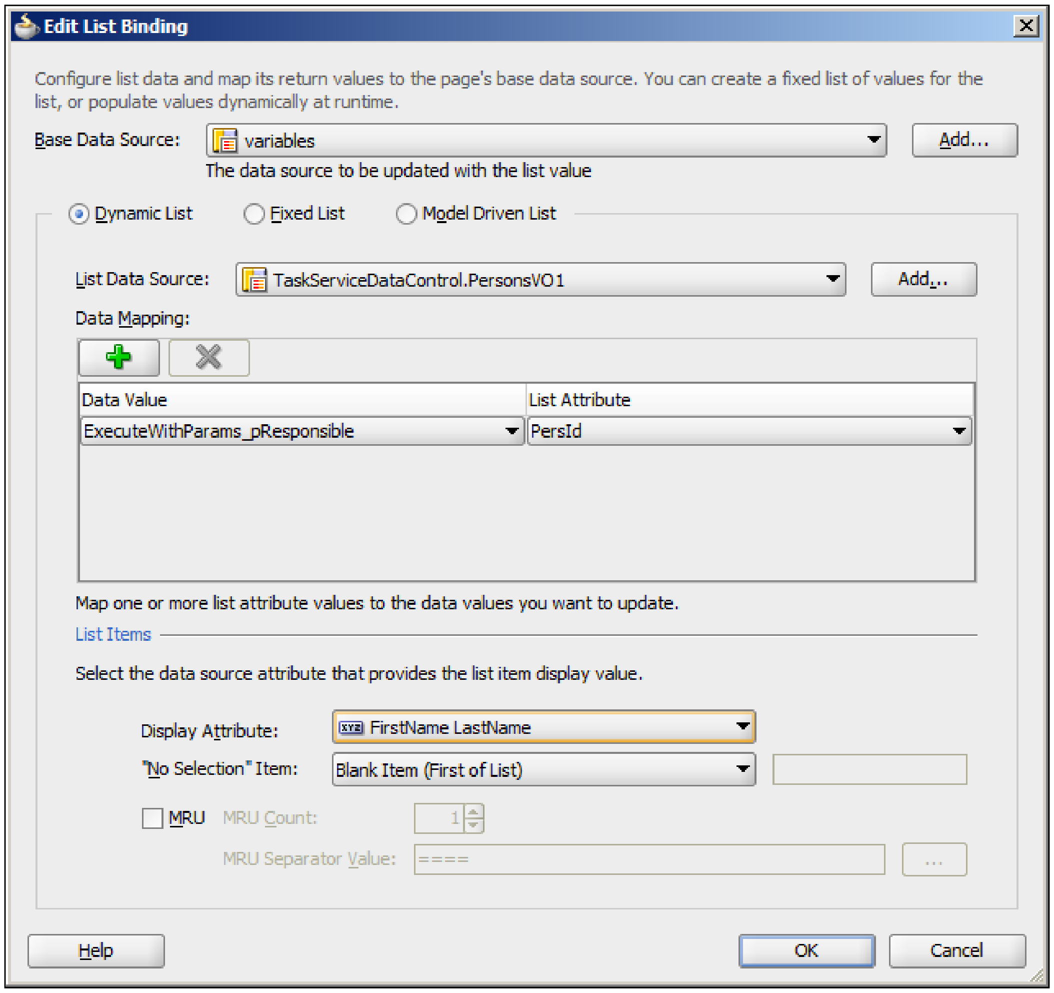
Referring back to the drawing of the tasks page, we can see that two groups of items are missing: The search criteria at the top and buttons at the bottom.

JDeveloper makes it very easy to create items that represent bind variables. If you expand the `AllTasksVO1` node, you will see all the attributes in the view object, as well as a node called **Operations**. If you expand the **Operations** node, you will see a number of standard operations that all view objects offer. One of these operations is **ExecuteWithParams**, and if you expand this fully, you will see the bind variables defined in the view object (`pResponsible`, `pProgramme` and `pText`):



To control the layout of the search criteria, first add a **Panel Group Layout** (from the **Layout Component Palette**) to the top facet.

The first criterion is the person responsible for the program. To add this criterion to the page, drop the `pResponsible` parameter onto the **Panel Group Layout** in the top facet. When you release it, a context menu appears. From this menu, select **Single Selection** and then **ADF Select One Choice**. The **Edit List Binding** dialog appears:



Edit List Binding

Configure list data and map its return values to the page's base data source. You can create a fixed list of values for the list, or populate values dynamically at runtime.

Base Data Source: variables

The data source to be updated with the list value

[✓ Dynamic List] Fixed List Model Driven List

List Data Source: TaskServiceDataControl.PersonsVO1

Data Mapping:

Data Value	List Attribute
ExecuteWithParams_pResponsible	PersId

Map one or more list attribute values to the data values you want to update.

List Items

Select the data source attribute that provides the list item display value.

Display Attribute: FirstName LastName

"No Selection" Item: Blank Item (First of List)

MRU MRU Count: 1

MRU Separator Value: ====

Leave **Base Data Source** at **variables** and click **Add** to add a new **List Data Source**. Choose the **Persons** view object as the source. Select `PersId` as **List Attribute** (the value that is bound to the variable). At the bottom of the dialog, choose **Initials** in the **Display Attribute** drop-down and set **"No Selection Item"** to **Blank Item (First of List)**, then click **OK**.

In the **Property Inspector** at the lower right, set the **Label** property for this drop-down to **Responsible**.

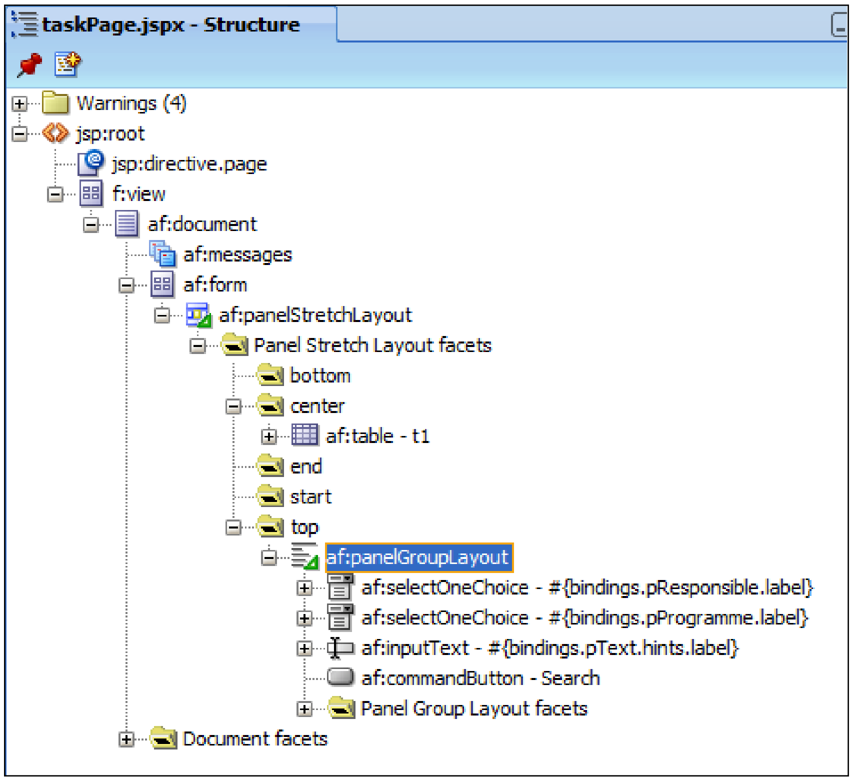
The second criterion is the name of the program. From the list of parameters (under **ExecuteWithParams**), drag the `pProgramme` parameter onto the **Panel Group Layout** next to the `pResponsible` drop-down and again drop as **Single Selection**, **ADF Select One Choice**. Again leave the **Base Data Source** unchanged and click on the **Add** button next to **List Data Source**. Choose the **Programme** view object as the data source for this drop-down list, and set **List Attribute** to `ProgId`. Then set **Display Attribute** to **Name** and again set **"No Selection" Item** to **Blank Item (First of List)**. Then click **OK**. In the **Property Inspector**, set the **Label** property to `Programme`.

The last criterion is the search text that is matched with the `TEXT` column. Drop the `pName` parameter next to the `pProgramme` parameter and this time drop it as **Text**, **ADF Input Text w/ Label**. Set the **Label** property to `Text`.

If you cannot see all the components on the screen, you can grab the bottom edge of the top facet where you added the items, and pull it down.

Finally, you need to create a button that actually executes the search. To achieve this, simply drag the **ExecuteWithParams** operation (the green gearwheel icon) onto the page inside the **Panel Group Layout**, next to the three search criteria. Because this is not a data element, but an operation, your drop choices are different. Choose **Operation**, **ADF Button**. The default text on the button is the name of the operation (`ExecuteWithParams`). In the **Property Inspector** panel in the bottom-right corner of the JDeveloper window, change the `Text` property to `Search`.

Your objects are probably placed below one another right now. To change this, select the panel group layout that you dropped in the top facet. It can be a bit hard to pick the exact right layout or input component in the design view of the page in the center of the JDeveloper window. Instead, look at the **Structure** panel at the bottom left of the JDeveloper window:


```
taskPage.jspx - Structure
[+] Warnings (4)
[-] jsp:root
    jsp:directive.page
    [-] f:view
        [-] af:document
            af:messages
            [-] af:form
                [-] af:panelStretchLayout
                    [-] Panel Stretch Layout facets
                        bottom
                        [-] center
                            [+] af:table - t1
                        end
                        start
                        [-] top
                            [-] af:panelGroupLayout
                                [+] af:selectOneChoice - #{bindings.pResponsible.label}
                                [+] af:selectOneChoice - #{bindings.pProgramme.label}
                                [+] af:inputText - #{bindings.pText.hints.label}
                                    af:commandButton - Search
                                [+] Panel Group Layout facets
            [+] Document facets
```

With the panel group layout selected, look at the **Property Inspector** in the bottom right-hand corner. Change the **Layout** property to **horizontal** to align the search criteria and the button horizontally. If you resized the top facet to see everything, you can make it smaller again.

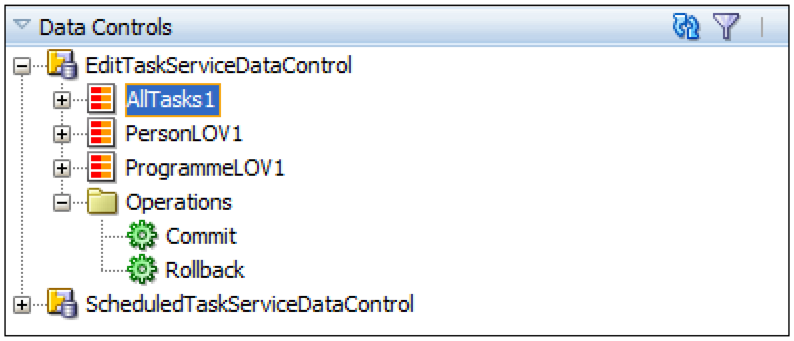
Running the Tasks Page with parameters

To make sure we got everything correct, let us run the page again. Right-click anywhere on the `taskPage` page and choose run from the pop-up menu. Because WebLogic is already started, your page should appear quicker this time.

Play around with the drop-down lists and try different values in the text search field. Each time you click on the search button, the table should update accordingly.

Adding database operations

We can now retrieve data from the database and edit it on the screen. However, we have not yet created a way to commit these changes to the database. For this, we will choose an operation at the application module level. The **ExecuteWithParams** operation belonged to the `AllTasksVO1` view object. But if you collapse all the view objects, you will see that the `EditTaskServiceDataControl` also has an **Operations** node with operations **Commit** and **Rollback**:



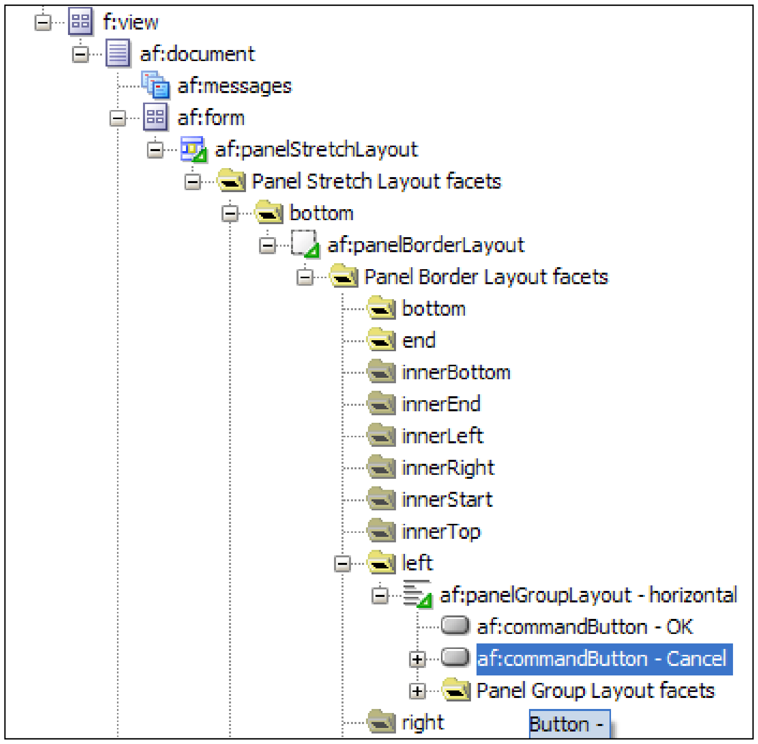
Before you drag these operations onto your page, drop a **Panel Border Layout** on the bottom facet of the page. If you refer back to the sketch of the user interface, you will see that we need some buttons (**OK** and **Cancel**) in the left side, and the **Timeline** button on the right. Because a Panel Border Layout has a number of facets along the edge, this is a good component to ensure this layout. However, since it does not offer a way to control orientation, we have another component to arrange the **OK** and **Cancel** buttons.

Use the Structure panel for arranging layout containers

When you have multiple containers within one another, drop them onto the structure panel at the lower left of the JDeveloper window. This part of the JDeveloper UI will present layout components in a tree structure, making it much easier to control where you drop components.

In the structure panel, expand the **af:PanelBorderLayout** component you just added, and drop a **Panel Group Layout** on the left facet. In the **Property Inspector** at the lower right in the JDeveloper window, set the **Layout** property of this **Panel Group Layout** to **horizontal**.

Then drag the **Commit** and **Rollback** operations from the **Operations** node of `EditTaskServiceDataControl` onto this **Panel Group Layout** and drop them as **ADF Buttons**. The **Structure Panel** should look as shown next. For both buttons, use the **Property Inspector** to set the text (to `OK` and `Cancel`), and delete the content of the **Disabled** field (under **Behavior**) to ensure that both buttons are always active:



```
f:view
  af:document
    af:messages
    af:form
      af:panelStretchLayout
        Panel Stretch Layout facets
          bottom
            af:panelBorderLayout
              Panel Border Layout facets
                bottom
                end
                innerBottom
                innerEnd
                innerLeft
                innerRight
                innerStart
                innerTop
                left
                  af:panelGroupLayout - horizontal
                    af:commandButton - OK
                    af:commandButton - Cancel
                    Panel Group Layout facets
                right          Button -
```

That is all there is to it—the ADF application module will automatically handle everything to ensure that your changes are either committed to the database or rolled back.

Running the tasks page with database operations

Run the page again, checking that your buttons are placed where you expect. Then make some changes to the data, and click **OK**. Use a database tool to verify that your changes are committed to the database, or close the browser and run the application again to verify that your changes are actually stored.

We will get back to the navigation button when we have built the other page.

Creating the scheduled tasks page

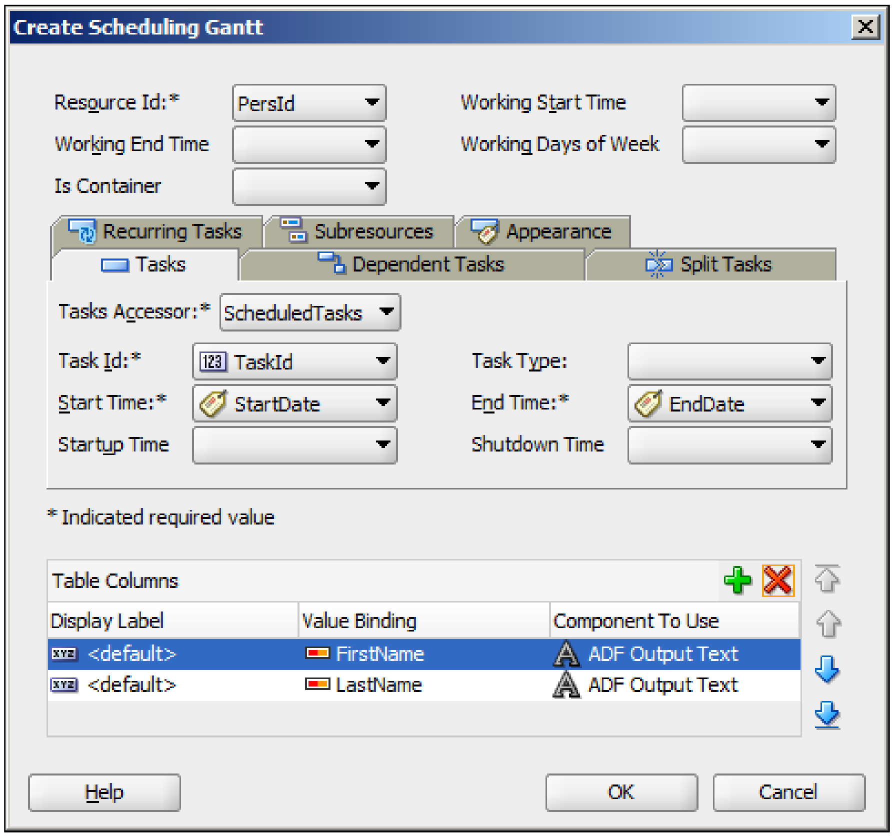
To create the scheduled tasks page, go back to the page flow. If you have closed the page flow window, you can find it again in the application navigator at the top-left in JDeveloper under **Web Content**, **Page Flows**. Double-click on the `schedulePage` icon and then **OK** to create the page.

Adding the Gantt component

Again, we start by dragging a **Panel Stretch Layout** component onto the page from the **Component** palette.

The component we need to implement the graphic representation of tasks assigned to persons is a Gantt chart of type Scheduling.

Under **Data Controls**, open the `ScheduledTaskServiceDataControl` node and drag the Persons view object onto the center facet and drop it as **Gantt** | **Scheduling**. The **Create Scheduling Gantt** dialog appears:


Create Scheduling Gantt

Resource Id:*	PersId	Working Start Time	
Working End Time		Working Days of Week	
Is Container			

Tasks Accessor:* ScheduledTasks

Task Id:*	TaskId	Task Type:	
Start Time:*	StartDate	End Time:*	EndDate
Startup Time		Shutdown Time	

* Indicated required value

Table Columns

Display Label	Value Binding	Component To Use
<default>	FirstName	ADF Output Text
<default>	LastName	ADF Output Text

Set the fields in this dialog as follows:

- **Resource Id: PersId**
- **Tasks Accessor: ScheduledTasks**
- **Task Id: TaskId**
- **Start Time: StartDate**
- **End Time: EndDate**

Under **Table Columns**, remove the extra columns, leaving only `FirstName` and `LastName`. Then click **OK**. You will see a graphical representation of a scheduling Gantt chart.

Click in the chart and then go to the Property Inspector to set values for the **StartTime** and **EndTime** properties (for example, `2011-10-01` and `2011-10-31`). The Gantt component does not automatically scale to the dates used, and making it do so involves a bit of code – we choose to leave this functionality out of this Proof of Concept.

Right-click anywhere on the page and choose **Run** to see the actual values in the browser, and play around with the capabilities of the Gantt chart component. We are using it in default configuration here, but there are many customization options. Refer to the help (Press *F1* with the Gantt component selected) or the documentation for a full description of this powerful component.

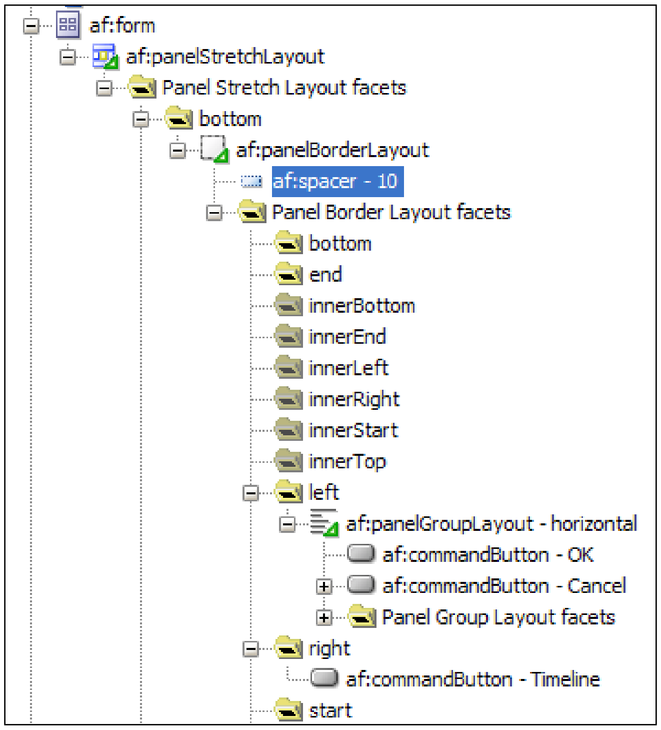
Navigation

The last thing we need to add to the Proof of Concept application is the navigation between the pages.

Open the `taskPage` and look in the **Structure Panel**. Find the facet called **Right** under the **af:panelBorderLayout** (right under the **af:panelGroupLayout** containing the `OK` and `Cancel` buttons). Drag a **Button** component from the **Component Palette** at the top right and drop it onto this facet. In the **Property Inspector**, set the **Text** property to `Timeline`, and under **Action**, select **goSchedule**.

The **goSchedule** option comes from the page flow—remember, this was the title of the only control flow arrow going away from the `taskPage`.

Finally, we need to drop a **Spacer** layout component from the **Component Palette** directly onto the **af:panelBorderLayout.** Your **Structure Panel** should now look like this:



```
af:form
  af:panelStretchLayout
    Panel Stretch Layout facets
      bottom
        af:panelBorderLayout
          af:spacer - 10
          Panel Border Layout facets
            bottom
            end
            innerBottom
            innerEnd
            innerLeft
            innerRight
            innerStart
            innerTop
            left
              af:panelGroupLayout - horizontal
                af:commandButton - OK
                af:commandButton - Cancel
                Panel Group Layout facets
            right
              af:commandButton - Timeline
            start
```

The reason we need the `spacer` component is that ADF automatically optimizes the page and does not show any facets that do not have content. So if the Panel Border Layout does not contain anything, the middle part is not shown, and the **Left** and **Right** facets are right next to each other. With the spacer in place, the central part will take up all the available space, pushing the **Left** facet left and the **Right** facet right.

Now open the `schedulePage`. Drag a **Panel Group Layout** onto the bottom facet and set the **Layout** property to **horizontal** and the **Halign** property to **right**. Then drop a **Button** onto the **Panel Group Layout**, set the **Text** to `Overview` and **Action** to `goTask`.

To test the navigation, you can now run the entire task flow. Open the `xdm-poc-flow` task flow, right-click anywhere on the page and click **Run**. Your application starts with the default activity (`taskPage`). Check that you can use the `Timeline` button to go to the `schedulePage`, and the **Overview** button to go back.

[The navigation between pages in the task flow only works if you run the task flow itself, not if you run the individual pages.]

Summary

In this chapter, we discussed what a Proof of Concept is, and why you need it. You got a very brief introduction to the ADF architecture, and saw or built a Proof of Concept application using the entire ADF technology stack, including the advanced Gantt chart component.

You are ready to go to your boss and demo what you can do with ADF. If he agrees that it seems like ADF is the right tool, your next step is to produce an overall design and estimate how long it will take to build the next generation of destination management software. This is the topic of *Chapter 2*, *Estimating the Effort*.

2
Estimating the Effort

You have convinced your boss that ADF has what it takes to build the next generation of destination management software. Now he is asking you how much this new enterprise application will cost, and how long it will take to build.

To be able to answer these questions, you will need to gather the requirements, do a high-level design of the solution, and estimate how long it will take.

Gathering requirements

The first step is to gather the requirements. This can be done in many different ways, depending on your organizational culture and environment:

- If you are subject to regulatory requirements (for example, in the aerospace or pharmaceutical business), you need very formal method
- If you are outsourcing development to an external supplier, you need exact requirements
- If development will be handled by an in-house IT department, you might get by with less formal requirements

At the formal end of the spectrum, you need a complete list of all requirements that you can test against. If your organization is used to a more informal approach, you might only produce a fairly complete list of use cases or user stories.

Know the requirements

Aim for as complete an understanding of the requirements as possible. Even if you are doing agile, iterative development, you still need to start out with a complete picture in your mind of the end result.

Use cases

Often, requirements are specified in the form of use cases. Each use case describes how to perform some task, so your complete specification will consist of multiple use cases. A small system might have less than 10 use cases while several dozen or even hundreds might be used to specify a large system.

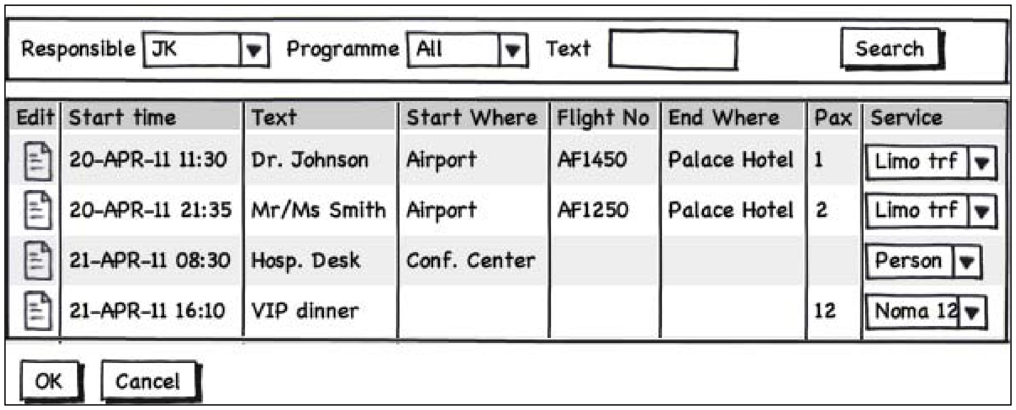
A use case describes the interaction between a person (by convention called an "actor") and the system. It describes what the system does for the user to provide business value to the actor, focusing on *what* is to be done and not on *how* it is to be done. It should therefore not contain any technical details about the implementation.

Use cases can be written at different levels of detail:

- A Brief use case will be just a few sentences to use in overviews and diagrams
- A Casual use case explains the use case in more detail, but still takes up less than a page of text
- A Fully dressed use case is a formal specification containing a number of fields such as Purpose, Summary, Actors, Normal flow, Exception flows, Business rules, and so on.

If you need a high degree of formality because of regulatory requirements or because the application is being developed by a supplier for a customer, you want fully dressed use cases. But if you are an in-house IT department building an application for internal use, you might start out with only brief or casual use cases and flesh out the details as the project progresses.

In the last chapter, we worked on Use Case 008 Task Overview and Edit. A more realistic version of the screen for this use case might look like this (with a link to a separate screen to edit all details):



Edit	Start time	Text	Start Where	Flight No	End Where	Pax	Service
	20-APR-11 11:30	Dr. Johnson	Airport	AF1450	Palace Hotel	1	Limo trf
	20-APR-11 21:35	Mr/Ms Smith	Airport	AF1250	Palace Hotel	2	Limo trf
	21-APR-11 08:30	Hosp. Desk	Conf. Center				Person
	21-APR-11 16:10	VIP dinner				12	Noma 12

In a formal description, this use case could look like this:

Name	Find and Edit tasks
Number	UC008
Version	1.2
Goal	To update task details if situation changes
Summary	Program responsible can search for tasks by responsible, program or text. Basic task attributes can be changed directly from overview screen; all attributes can be changed in a pop-up detail screen.
Actors	Program responsible
Preconditions	A program has been defined
Trigger	It becomes necessary to change task details, for example, because flights or number of passengers change
Primary flow	1. User selects responsible person, program or a free text or a combination of these and initiate a search 2. System shows only the tasks that meet all conditions (free text matched anywhere in Text field). Attributes show include Start Time, Text, Start Where, Flight no., End Where, Pax and Service 3. User performs necessary changes 4. User approves or cancels the changes
Alternative flows	3a. User clicks on edit button next to task 4a. System shows a detail screen with all task attributes 5a. User performs necessary changes 6a. User closes detail window Rejoin main flow at 4.
Post conditions	All tasks remain valid
Business Rules	BR024 Task start time after program start time BR025 Valid flight number BR026 Task pax <= service max pax
Notes	Adding new tasks is out of scope for this use case
Author and date	Sten Vesterli, 2011-03-26

User stories

Agile software development methods tend to find fully dressed use cases too heavy and prefer to work with shorter descriptions, usually called user stories. By tradition, these are supposed to be so short that they can be fit on a single 3 x 5 index card. In principle, there is no difference between a casual use case and a user story.

User stories are only useful as requirements if supplemented with some kind of acceptance test description, establishing a common understanding between the business user and the developer about what constitutes a successful implementation.

Critical success factor: The User

A requirement for success with agile development is that the business users are always available to answer questions throughout the project. Unless you have solid buy-in to this process from your business users, agile development is unlikely to succeed.

Non-functional requirements

It is normal in a software development project to focus mainly on the functional aspects—what should the system **do**. However, there are many other aspects to software quality; for example, ISO-9126 defines the following aspects:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

It is important that non-functional requirements are documented in a measurable and testable way, just like the functional requirements. Response time requirements need to be specified in seconds (or fractions of seconds) for a specific function:

Requirement	Description
NFR002	Time from user initiates search until filtered data is shown on the screen is < 0.5 seconds.

Performance/capacity requirements need to specify the exact load the system is expected to handle (for example, 50 concurrent users sending a request every second, receiving a response within 0.5 second).

One common non-functional requirement is that the system should be "user-friendly". This is a very inexact requirement and needs to be detailed into something measurable. For example: "An untrained user is able to enter five expense report items and submit the expense report in less than three minutes."

Requirements list

If you are working with formal requirements, you need to collect all the requirements in a common list where each requirement is given a unique ID. When you are building your test cases to prove that the application works as specified, you can map the test cases to the requirements. If all requirements are covered by a test case, and all test cases succeed, your application is complete.

Ensuring complete test coverage of the requirements for an enterprise application is a major undertaking, typically requiring a test management professional.

Screen design

In many enterprise applications, there are a small number of central screens where most of the day-to-day work is being done. Additionally, there might be screens with specific layout and design criteria, for example graphic dashboard-style screens. If you have produced a use case list, you should be able to identify these special screens easily; you might even have built some of these as part of your proof of concept.

For these screens, it makes sense to specify the screen layout in some detail as part of the requirements process. With 100+ ADF components available, it is normally a good idea to select a limited set of components and design all screens using this limited set. This provides a more uniform user interface and makes it easier for the development team to build. This selected set should be documented to the users so they know what behavior to expect from a drop-down list box, data selection, and so on.

Because the UI design part is actually a rather small part of the overall development time in an ADF application, and because the UI tends to evolve over the lifetime of the project, it is not cost efficient to produce a detailed layout for every screen. Typical data maintenance screens will be totally acceptable in the default layout achieved when dropping a data control on to a page, for example, an ADF Form.

You should design these screen layouts (sometimes known as "wireframes") in a tool that makes it easy to make changes. You can use a specialized tool for wireframe screens like Balsamiq (`http://balsamiq.com`), which deliberately produces a "hand-written" look to make it clear to users that this is just a sketch and can be freely changed until exactly the right design has been achieved. The sketches in *Chapter 1, The ADF Proof of Concept* and earlier in this chapter have been produced with the Balsamiq tool.

An alternative is to produce high-fidelity prototypes using a tool like Microsoft Visio. Oracle has made a set of Visio Stencils available on `http://samplecode.oracle.com` that you can download and use.

Be careful about using these high-fidelity prototypes (or actual JSF pages) as requirements; you might run into several problems:

- It enables the misconception that the application is almost done
- It tends to fixate a very specific image in your user's minds – if an alternative design idea appears during development, it can be hard to deviate from the almost-finished screen the user believes to have seen
- You are more likely to get bogged down in detailed discussions about colors and fonts in a project phase where the focus should be on the functionality

Application architecture

When you know the requirements and have screen design for the most important screens, you can start designing the solution. This does not mean that you need to know every piece of code you're going to write, but you do need to be able to break down the work in manageable chunks.

The Work Breakdown Structure

This is typically done in the form of a hierarchical **Work Breakdown Structure** (**WBS**) that decomposes the entire application development effort into a number of **work packages**. The following list contains the work packages you will probably need when building an enterprise ADF application. Many of the items listed here haven't been explained yet, but will be covered in subsequent chapters—they will make sense when you return to this chapter after reading the rest of the book.

Your Work Breakdown Structure is likely to include:

- **Technical design**

  Detailed design documents providing any information the programmer will need and which is not already in the requirements.
- **Server setup**
  - You need both a development and a test environment with a WebLogic application server and a database
  - You need a source control repository (for example, Subversion) if you do not already have one

    - You need an issue tracking tool (for example, Jira) if you do not already have one
    - You need a development Wiki if you do not already have one

- **Development workstation setup**

  Getting each development workstation set up with JDeveloper and a local database + connection to source control and issue tracking.

- **Development handbook**

  Everything a new developer needs to know to work on your project. Includes how to structure workspaces and project, how to work with the database, how you intend to do version control, build management, configuration management, issue management. Naming conventions. Security strategy, test strategy, UI guidelines. HOWTOs for any common development tasks (for example, how to display an error, how to create an ADF library, how to write localizable strings).

- **Prototyping**

  Creating standard ways of handling common functionality. To be documented in HOWTO sections in the development handbook. Possibly multiple work packages for different topics.

- **Framework extension classes**

  Your own base classes that all your ADF business components should be based on.

- **Data model**

  If you do not already have a data model, you need to create tables and other database objects.

- **Entity objects for all tables**

  If you are working directly with relational tables, this should be a small task. Include any special data access coding, if not working directly with relational tables.

- **View objects for common value lists**

  Including a common application module.

- **Graphical design**

  If you are designing a public-facing application, you might have very strict graphical design guidelines. This is likely to take less effort for internal applications. Output from the graphical design should be example HTML pages.

- **Skinning and templates**

  Creating page templates and a "skin" for your application, defining its visual identity.

- **Usability Testing**

  A developer cannot guess what a user finds easy. Usability testing is a specialty of its own, requiring trained professionals.

- One work package per use case, subdivided into the following:
  - Task flow (typically one for each use case)
  - Data model for each screen (view objects and view links)
  - Screens (JSF page fragments)
  - Test cases
  - Technical documentation

How much testing?

It will make a big difference to your estimates how you plan to test your use cases. An explorative test where the developer plays around with the screen following a short checklist does not take much time, but recording a re-playable user interface test does. Decide on this before you estimate. We will return to testing in a later chapter.

- **Business Logic packages**

  Depending on your requirements, this can be anything from one small work package to a collection of large ones.

- **Integration packages**

  Depending on your integration needs. Typically one work package per interface (web service, file load, and so on).

- **Main application**

  This work package will collect all the bounded task flows from the individual use cases into the final application.

- **Automated build procedure**

  Setting up Ant, Maven or a similar tool to automatically build your completed application directly from source in your version control repository.

- **System Integration Testing**

  Reserve at least 10% of your total development time for integration testing. This is testing the whole application—testing individual components should happen in the use case work packages.

- **Coordination and project management**

  For example, 5% of the total development effort in coordination time – remember, a two-hour project meeting is already 5% of a work week. For project management, use, for example, 20% of the total development effort (but typically not more than one full-time resource).

Estimating the solution

With your work breakdown structure in hand, you can start the work of estimating the real work involved in each group of tasks. Estimate the *effort* needed to perform the task in hours or days (measured in **ideal engineering hours**, assuming concentrated, uninterrupted work on the task). Do not fall into the trap of estimating in *duration* —duration estimates will vary wildly, depending on how much non-project work the person doing the estimate expects to be doing at the same time. .

Use small tasks

If you find that a work package has an estimate of more than 80 hours, revisit the work breakdown structure and split the task into smaller sub-tasks. An estimate of 80+ hours very often indicates an incomplete understanding of the task and carries a large risk of overrunning the estimate.

The individual use case work packages above might break the 80-hour limit and are, therefore, divided into sub-packages.

Top-down estimate

If you are the project manager, you can probably produce a rough estimate of the total effort involved in the project. For this, you rely on your experience with similar projects and your intuition.

Some project managers do not like the idea of using "intuition" because it does not feel scientific and exact. However, your other-than-conscious mind can process a lot of information and will be able to produce some quality input to your estimation process.

Of course, you do not start a multi-man-year project based solely on intuition—you combine the top-down estimate with a bottom-up estimate.

Bottom-up estimate

In order to produce the bottom-up estimate, you ask people capable of performing each task how much effort (in hours or days) it will take to produce the necessary output. Some projects prefer to let several people do independent estimates, while other project methodologies like Scrum prefer team estimates using collaborative techniques like "planning poker" (see `http://www.planningpoker.com`).

The productivity you observed during the proof of concept will give you an idea of the effort involved in some of the common ADF development tasks.

You need to make clear what you include in the estimate—developers typically forget to include things like technical documentation and repeatable test cases.

The rest of this section describes an estimation technique often used for formal estimates that go into agreements and contracts—for example, when a system integrator is making an offer to a customer to build an application according to agreed specifications. If you are an IT department building an application for internal use, Agile methods like Scrum where development is done in a number of fixed time windows (called "sprints") might fit your needs better.

Three-point estimates

If you ask someone for just one estimate, it is natural that the estimate will be padded with a bit of buffer time. All sorts of unforeseen complication may emerge, so a developer will try to anticipate these and include them in his or her estimate.

Unfortunately, even if complications do *not* occur, task still tend to take the estimated time.

In order to respect the uncertainty of the task without padding every task with buffer time for worst-case scenarios, developers and others producing estimates should produce a **three-point estimate** for each task:

- An **optimistic** estimate: The best case, if things are easier than expected. You stumble upon a framework class that can do the task; a wizard in JDeveloper generates code for you, and so on.
- A **likely** estimate: The time you realistically expect the task to take.
- A **pessimistic** estimate: The worst-case scenario, if things are harder than expected. The class you thought you could use does not do exactly what you want, you run into a baffling bug when testing with multiple users, and so on.

  Do not allow the pessimism to take overhand; you do not need the pessimistic estimate to include tornadoes destroying the test server.

Three-point estimates like these will clearly show the project manager which tasks are not clearly specified or carry a greater risk. If the pessimistic estimate is much higher than the likely estimate, the person doing the estimate is unsure about either the task or the means to complete it. This can be addressed by specifying the task in greater detail and possibly performing a short, dedicated proof of concept to allay any doubts the developer or other estimator has about the task.

The data set from three-point estimates can also be used to calculate an **expected time** according to the formula in Program Evaluation and Review Technique (PERT). This technique was developed by very clever people building Polaris nuclear submarines in the 1950s and has been used since.

The expected time is calculated as follows:

$$t_{expected} = (t_{optimistic} + 4 \times t_{likely} + t_{pessimistic}) / 6$$

This time takes the uncertainty into account and produces a better number than just using the most likely time. The three points will also be used for some clever math when we get to the end of the chapter where we'll be adding everything up.

Grouping: simple, normal, hard

Your work breakdown structure is likely to include many tasks of similar complexity. Naturally, you are not going to estimate each of these individually, but rather estimate an average complexity for the whole group and simply multiply this effort with the number of code modules in the group to arrive at a total estimate.

For an ADF application, you might use the following grouping:

Element	**Simple**	**Normal**	**Hard**
Entity objects	Based on relational tables	Based on API offering insert/update/delete functions (for example, a PL/SQL API)	Based on API that does not map directly to insert/update/delete operations
Value list	Based on relational table	Dependent value lists (content depends on other selections)	
Data model	Data maintenance, 1 view object	Average screen, 2-5 view objects	> 5 view objects
Task flow	1-3 pages	4-10 pages with simple flows	> 10 pages or complex flows

Element	Simple	Normal	Hard
Screen	Simple data maintenance on one view object	Normal screens based on 2-5 view objects, no special components	Based on > 5 view objects or using complex components (for example, tree, visualization, drag-and-drop)

You can use more groups if your application has a wider variety of tasks, or you might estimate a few hard tasks individually if they are of a higher complexity than your other "hard" tasks.

More input, better estimates

Research shows that the average of independent estimates from several people is likely to be closer to the correct value than any one estimate. Many Agile development methods also recommend to involve developers, architect, testers, and customer in discussions of the estimate.

You can try this in the office: Ask your colleagues what the distance is between two well-known cities some distance away. You will find that the average of just five people will be pretty close to the real distance even if some of the estimates are way off.

You do not need multiple estimates for all of the items in your work breakdown structure, but for critical tasks or tasks where the worst-case estimate is far from the likely estimate, consider getting a second, third, or fourth opinion.

The following table shows examples of what estimates could look like for a few of the items from the Work Breakdown Structure described earlier. All estimates are in ideal engineering hours:

Task		Likely	Pessimistic
Server setup			
Development Server	8	16	24
Subversion, Jira and Wiki	4	10	20
Entity Objects (55 tables)	11	27	55
Graphical Design	20	40	60
Skinning and Templates			
Skin	4	10	40
Page template incl. menu	8	16	24
Usability testing	40	80	160
UC 008 Task Overview and Edit			

Task	Optimistic	Likely	Pessimistic
Task Flow	1	2	3
View objects	1	3	5
Overview screen	2	4	8
Detail screen	3	6	12
Test cases	2	4	6
Technical documentation	1	2	3

You will notice that some tasks have larger variation than others. For example, usability testing is likely to take 80 hours, but might take up to 160 hours. The reason for the high pessimistic estimate is that we might need several iterations before the usability is as good as we want. For other tasks, the optimistic, likely, and pessimistic values are very close together. This indicates well-defined, low-risk tasks we are pretty sure how to complete.

Adding it all up: the final estimate

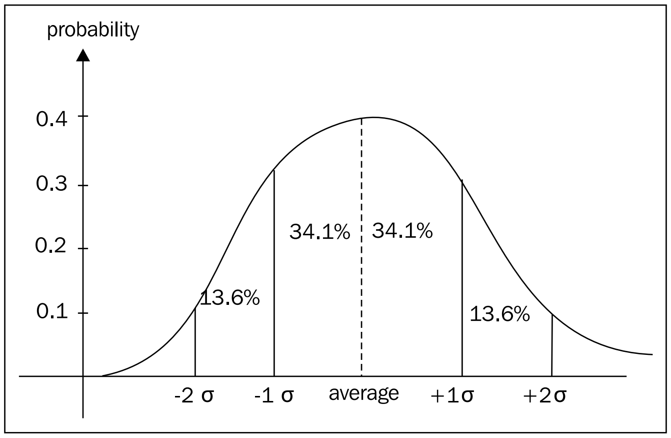
When you have gathered all the detailed task estimates, you need to add up the details to a total estimate for the entire project.

As a starting point, you add up all the expected task efforts. Remember that these are calculated based on your three-point estimates using the formula earlier in this chapter. This total is the most likely total effort needed to complete the project.

Swings and roundabouts

A fairground owner will say: "What you lose on the swings, you gain on the roundabouts". A developer will recognize this: Some things take longer (closer to the pessimistic estimate), and some take shorter (closer to the optimistic estimate). However, it is *extremely unlikely* that everything takes as long as the pessimistic estimate—just as it is extremely unlikely that everything goes swimmingly according to the optimistic estimate.

A statistician will illustrate this fact with a **normal distribution curve** showing probability or likelihood on the vertical axis and project effort on the horizontal axis. This bell-shaped curve is high in the middle, indicating that it is most likely that your total project effort will be somewhere near the sum of all the expected task efforts. It drops off towards the ends, showing how likely or unlikely it is that your project duration will be dramatically different from the middle value:



The shape of the curve is defined by what is known as the **standard deviation** that can be calculated from your three-point estimates. There is some math behind it, but the net result is that it is 95% likely that your total project effort falls within plus/minus two standard deviations. (If you're interested in the math, there is a good treatment on Wikipedia: `http://en.wikipedia.org/wiki/Standard_deviation`.)

Calculating standard deviation for a task

You can calculate the standard deviation for each task using your three-point estimates. Simply use a spreadsheet program like Microsoft Excel and the standard deviation function (`STDEVP` in Excel). The inputs to this function should be six figures (one optimistic, four likely, and one pessimistic) just like we used when calculating expected time for a task.

This calculation will show a large standard deviation (and greater uncertainty) if your optimistic and pessimistic values are far from the likely value.

Calculating standard deviation for a project

To get from the standard deviation of the individual tasks to the standard deviation for the whole project involves a bit of math. What you do is you calculate the square of the standard deviation (also called the variance), and then add up the variance values for each task. You then take the square root of the sum of variances to get the standard deviation for the entire project.

Remember from above that it was 95% likely that the actual value is within plus/minus two standard deviations from the middle value? So if your most likely total effort is 1450 hours and the calculated standard deviation for the entire project is 150, you can tell the business: "We expect the effort of the project to be 1450 hours with a 95% probability of falling in the interval from 1150 to 1750 hours."

The following table illustrates how to calculate a final estimate. Because the entire estimate would take up several pages, this calculation example uses just four tasks from the work breakdown structure.

Task	Optimistic	Likely	Pessimistic	Expected	Std.dev.	Variance
Development Server	8	16	24	16	5	21
Page templates	4	10	20	11	5	22
Graphical design	20	40	60	40	12	133
Usability testing	40	80	160	87	36	1289
Total				153		1466
Proj. std. dev				38		

The values in the **Expected** column are calculated according to the formula earlier in this chapter. In Excel, it would look something like `(B2+4*C2+D2)/6`.

Your spreadsheet software can calculate the values in the **Standard Deviation** column. In Excel, this would look like `STDEVP(B2;C2;C2;C2;C2;D2)`. This calculates the standard deviation of one optimistic value, four likely values and one pessimistic value.

The values in the **Variance** column are simply the square of the standard deviation. In Excel, this could look like `POWER(F2;2)`.

Finally, you calculate the sum of all variance values and take the square root of this sum to get the standard deviation for the whole project. A statistician can explain to you why you cannot just add up standard deviations. In Excel, this calculation would look like `SQRT(G6)`.

With a total expected time of 153 hours and a standard deviation of 38 hours, you get:

- Expected project effort is 153 hours
- It is 95 percent likely that the project effort will fall between 77 and 230 hours (plus/minus 2 times the standard deviation)

If this is too wide a span for the business to sign off on, you need to address the major uncertainties. If, for instance, you were to limit usability testing to a maximum of 100 hours, the expected time for this task drops to 77 hours and the project standard deviation to 22. This leads to an expected project effort of 143 hours with a 95% confidence interval of 99 to 188 hours.

Again, this estimation methodology is intended for formal estimates where you need to agree on project effort up front. If you do not need to commit to a fixed time and cost at the beginning of the project, Agile development methods might fit your needs better.

Sanity check

Once you add up all your bottom-up estimates, you should arrive at a total close to the project manager's top-down estimate. If the estimates are not fairly close, your project contains some uncertainty that you need to examine.

The project manager might find that the bottom-up estimate is higher because it includes tasks he or she did not consider in the top-down estimate. That is fine. But if you have a major discrepancy and cannot find the reason, you need to revisit your estimates. As described above, you get better estimates if you let more people do the estimation and then calculate averages. Do this for both your top-down and bottom-up estimates until the total bottom-up estimate is approximately the same and the top-down estimate.

From effort to calendar time

Remember that we have been discussing *effort* in this chapter, calculating in ideal engineering hours. You need to convert this into actual calendar time, taking into account vacation, illness, training, support of existing systems, tasks for other projects, company meetings, and many other things.

If you have already implemented detailed time tracking in your organization, you can get a good idea of the development efficiency of each developer from historical data. If you do not have this data, start your project plan assuming a 50% efficiency for everyone. Then follow up on how many hours are actually spent on development tasks for each person. This can be very different from person to person because of the varying other tasks each team member will have.

Summary

You have gathered all the requirements, created an initial application architecture, and a Work Breakdown Structure for the XDM project. Together with the other developers in DMC Solutions, you have created three-point estimates for each task in your project and calculated the total effort needed to build DMC Solutions' next generation destination management.

Your boss was impressed with your detailed estimate and liked the fact that you had calculated a 95 percent probability worst-case value. He has now gone to the CEO for funding – if he gets the go-ahead, your next task is to get the project team organized. That is the topic of *Chapter 3*, *Getting Organized*.

3
Getting Organized

You have proved that ADF is the right tool for your enterprise project. You have estimated how long time it will take to build the application, and the business has approved the project.

Time for some programmers to get together and start coding, right?

Wrong. If you are to build a successful enterprise project, you need to think about the skills and people you need before you start. In addition, you need to set up a few tools and establish some guidelines to ensure that everyone on the team will be working efficiently together to achieve the common goal.

Skills required

For a small application, you might have to do all the work yourself. If you have ever built a whole application, you might remember that some parts were fun and others less fun. And, while functional, your application probably did not win any prizes for design or usability.

If your enterprise application is going to be a success, people should *want* to use it. That means it has to be visually attractive and user friendly—public sites such as Facebook, Flickr, and the various Google applications have set the bar fairly high. Your users *expect* the enterprise application you are building to live up the sites they see and use regularly.

These applications are not built by one person. They are large (like your application), so they need a sizable team of programmers. But they are also complete—the visual identity and usability is as much part of the overall experience as the code. If you are to live up to these high demands, you need a team with many different skills, including the following:

- ADF framework knowledge

- Object-oriented programming
- Java programming
- Database design and programming
- XML
- Web Technologies (HTML, CSS, JavaScript)
- Regular Expressions
- Graphics design
- Usability
- Testing

We will examine these necessary skills in more details in the following sections.

ADF framework knowledge

Everybody on the team will need basic knowledge of the ADF framework. This includes project managers, testers, graphics designers, and usability experts too. Having a common understanding of what the framework can and cannot do will make communication within the team much easier.

The programmers will, of course, need a deeper understanding of the framework, but not everybody has to be an expert. An experienced programmer should just need a basic one-week training class and a couple of weeks work under experienced supervision in order to be productive with ADF.

Finally, you need at least one person with a deep understanding of how ADF works. This person will define project standards and provide guidelines for how to use ADF effectively. The same functionality can be implemented in many ways in ADF. It is the task of the ADF expert to ensure that you use the framework as much as possible and do not code things that ADF can handle declaratively.

If you do not have anyone in your organization with these skills, you should seriously consider hiring an experienced consultant to help you get your project off the ground in the right direction. Once you have built up some level of skill with ADF, you might get by with a part-time expert as long as that person is on call by phone or e-mail to prevent you from getting stuck or travelling too far down a wrong path.

Work with the Grain

When a carpenter picks up a planer to shave a piece of wood, he will be working with the grain of the wood. This allows him to produce a smooth surface with a minimum of effort. If he works against the grain, the plane iron will lift the wood fibers. This makes it harder to push the plane, and the surface will be rough.

Similarly, you need to work with the grain of the ADF framework in order to produce a good-looking, efficient enterprise application with the minimum amount of effort.

Object-oriented programming

You also need someone on the team who understands the principles of object-oriented programming, but again, not every programmer on your team needs this skill.

The ADF framework takes care of most of the heavy lifting you need to build a Java application on top of a relational database. But you will need to build your own layer of framework extension classes that fit into the Java object hierarchy between the standard framework classes supplied by Oracle and the ADF Java objects you are building in your application. You might also occasionally need to develop support classes in Java, and because Java is an Object-Oriented programming language, some OO skills are necessary in the team.

Java programming

Not everybody who writes needs the skills of Shakespeare. But everybody who writes need to follow rules of spelling and grammar in order to make themselves understood.

All serious frameworks provide some way for a programmer to add logic and functionality beyond what the framework offers. In the case of the ADF framework, this is done by writing Java code. Therefore, every programmer on the project needs to know Java as a programming language and to be able to write syntactically correct Java code. But this is a simple skill for everyone familiar with a programming language. You need to know that Java uses { curly brackets } for code blocks instead of BEGIN-END, you need to know the syntax for if-then-else, constructs and how to build a loop and work with an array.

But not everyone who writes Java code needs to be a virtuoso with full command of inheritance, interfaces, and inner classes.

Database design and programming

In a modern three-tier application, you have two servers at your disposal: The application server and the database server. You can build your entire application using just the ADF Framework and the application server. But if you are running your application on top of an Oracle database, why not make full use of the features the database offers?

Some tasks are better handled in the database, for example, if you need to process many rows according to some rule, without needing any input from the user. It is inefficient to pull all data to the application server, process it, and push it back to the database. If you program stored procedures in the database (using PL/SQL in the case of the Oracle database), your application server simply needs to tell the database to start working and can then return to handling the interaction with the user.

It will therefore be an advantage to have someone on your team with database programming skills.

Additionally, in case you are building a brand-new enterprise application that does not have to work with existing data and tables, you need someone on your team who understands database design. The database design is the foundation your enterprise application stands on. You will have a very hard time building a robust enterprise application if the foundation is not rock solid.

XML

The ADF framework is meta-data driven. This means that most of the application is not actually programmed in a programming language like Java, but is instead defined through JDeveloper. These definitions are stored in the form of XML files. You will notice that, for example, business components have a **Source** tab, allowing you to see the raw XML file.

You do not have to write XML files to use ADF, but it will be an advantage to know a little bit about XML so you can read the files JDeveloper builds.

Powerful and dangerous

JDeveloper also allows you to actually edit the XML files directly. This is not necessary during normal development—you can perform almost all changes through the wizards and dialogs in JDeveloper. An experienced ADF developer might sometimes edit the XML directly; this is a powerful, but also dangerous approach that can wreck your application, or worse, introduce subtle, very hard-to-find errors. As a beginning ADF developer, do not edit the XML.

Web technologies

When your ADF application is running, the end user is interacting with a web page in a browser. This means that your application must use the web technologies the browser understands: HTML, Cascading Style Sheets (CSS), and JavaScript. The ADF framework takes care of most of the details for you, but it is good to have someone on your team that understands these technologies. That person can both help you understand any limitations you might encounter and how to work around them.

Regular expressions

JDeveloper allows you to use **regular expressions** to define validation rules. Regular expressions are arcane, almost magical constructs that can express complex requirements in very compact form. For example, the following regular expression can be used to validate an e-mail address:

```
[A-Z0-9._%-]+@[A-Z0-9.-]+\.[A-Z]{2,4}
```

This is clearly much shorter than writing a block of Java code, but unfortunately it is also impossible to read for someone who does not know regular expression syntax.

If you do not want to learn regular expressions, that's OK. But in an enterprise application, you'll probably be defining so many special validation rules that it will be worth the effort for someone to learn this syntax.

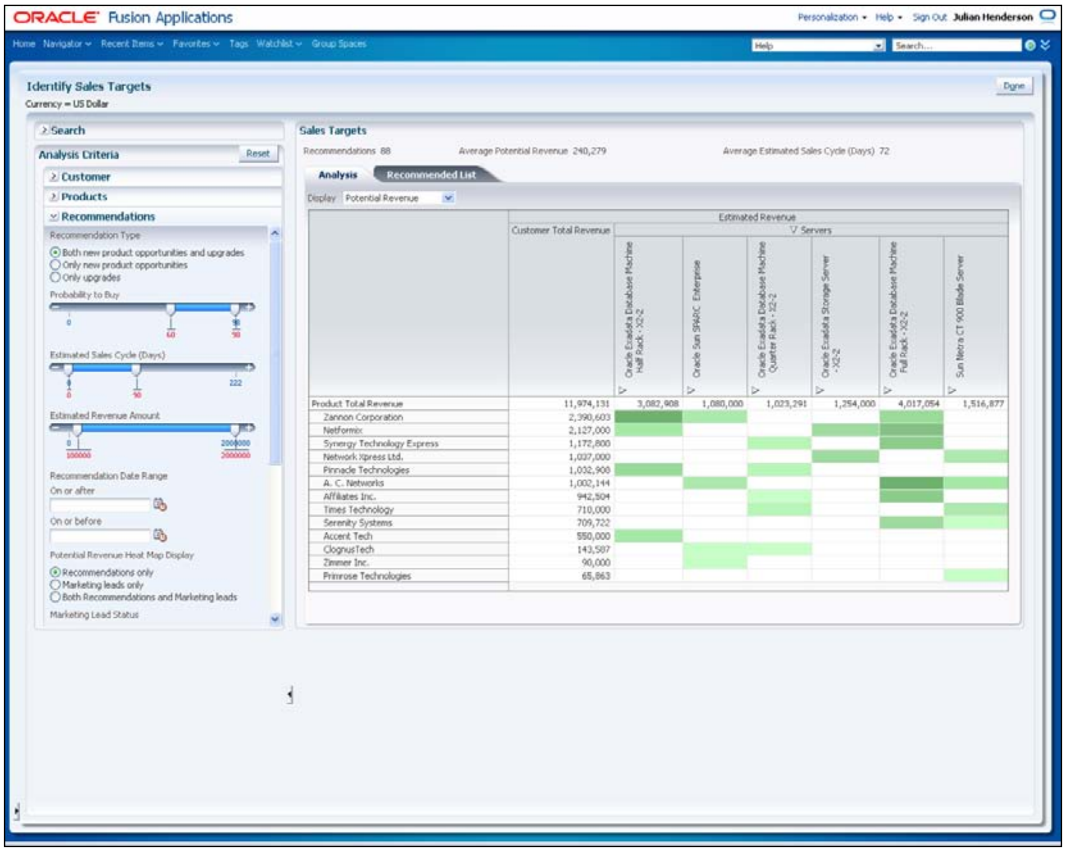
In case you are interested, the above regex reads:

Any number of characters A-Z, digits 0-9, periods, underscores, percent or dash, followed by an @, followed by any number of characters A-Z, digits 0-9, period or dash, followed by a period, followed by 2 to 4 characters A-Z. (Our example assumes your regex parser is in case-insensitive mode.)

Graphics design

Have you checked Facebook today? Or LinkedIn, or Orkut, or Flickr, or last.fm? Have you been on amazon.com or eBay? Do these sites look like the enterprise applications you use at work?

Probably not, unless you are running Oracle Fusion Applications as shown in the screenshot by Oracle:


ORACLE Fusion Applications
Personalization ▾ Help ▾ Sign Out Julian Henderson
Home Navigator ▾ Recent Items ▾ Favorites ▾ Tags Watchlist ▾ Group Spaces
Help
Search...
Done
Search
Reset
Customer
Products
Recommendations
Analysis
Recommended List

Identify Sales Targets

Currency = US Dollar

Analysis Criteria

Recommendation Type: [✓ Both new product opportunities and upgrades] Only new product opportunities Only upgrades

Probability to Buy: 0 60 90

Estimated Sales Cycle (Days): 0 90 222

Estimated Revenue Amount: 0 100000 2000000

Recommendation Date Range: On or after / On or before

Potential Revenue Heat Map Display: [✓ Recommendations only] Marketing leads only Both Recommendations and Marketing leads

Marketing Lead Status

Sales Targets

Recommendations 88 Average Potential Revenue 240,279 Average Estimated Sales Cycle (Days) 72

Display: Potential Revenue

		Estimated Revenue					
	Customer Total Revenue	Servers					
		Oracle Exadata Database Machine Half Rack - X2-2	Oracle Sun SPARC Enterprise	Oracle Exadata Database Machine Quarter Rack - X2-2	Oracle Exadata Storage Server - X2-2	Oracle Exadata Database Machine Full Rack - X2-2	Sun Netra CT 900 Blade Server
Product Total Revenue	11,974,131	3,082,908	1,080,000	1,023,291	1,254,000	4,017,054	1,516,877
Zannon Corporation	2,390,603	🟩	🟩			🟩	
Netformix	2,127,000	🟩			🟩	🟩	
Synergy Technology Express	1,172,800			🟩		🟩	
Network Xpress Ltd.	1,037,000				🟩		🟩
Pinnacle Technologies	1,032,900	🟩		🟩			
A. C. Networks	1,002,144		🟩			🟩	🟩
Affiliates Inc.	942,504			🟩		🟩	
Times Technology	710,000			🟩			🟩
Serenity Systems	709,722					🟩	🟩
Accent Tech	550,000	🟩					
ClognusTech	143,507		🟩	🟩			
Zimmer Inc.	90,000		🟩				
Primrose Technologies	65,863						🟩

Your users experience modern web applications every day, and expect your enterprise application to be as visually attractive and user friendly. Is the average programmer able to produce such an experience? Left alone, probably not. That is why you should have someone with graphics design skills on your team.

Usability

Speaking of Facebook, did you take a Facebook class? No, you did not. Modern web applications are built with such a focus on usability that everyone can use them right away. Your users are expecting the same thing from you: That the enterprise application you build is so user-friendly that they can use it without taking training classes.

That is not at all easy. Programmers tend to be people with a specific way of thinking, and without guidance, they build applications for people like themselves. The purpose of usability studies is to ensure that applications are built for the people who will actually be using them.

Usability experts will use a variety of tools, from low-tech paper prototypes to high-tech eye tracking equipment to achieve this goal.

Testing

When the original Amazon.com website went into beta testing, it had of course already been tested thoroughly by the developers. But Jonathan Leblang, one of the beta testers, immediately found that you could order a negative number of books—meaning that Amazon owed you money and would refund it to your credit card.

This anecdote illustrates that developers typically make poor testers. They have full knowledge of all the things the application is supposed to do, so they tend to test only a narrow range of variations on the normal case. A professional tester, on the other hand, will test the entire range of possible inputs (including negative numbers).

Organizing the team

A complete enterprise application development project team needs to fill the following roles:

- Project manager
- Software architect
- Lead programmer
- Regular programmer
- Build/configuration manager
- Database and application server administrator
- Graphics designer
- Usability expert
- Quality assurance
- Test manager and tester

Additionally, if you are building an application from scratch or are making significant changes to an existing application, you will also need a data modeler.

This does not mean that your team has to have a dozen people, if your enterprise application is not very big, you can get by with fewer. But you do need to fill all of these roles, one person can often fill more than one role.

Project manager

Naturally, you need a project manager to run an enterprise project. Project management is a well-documented discipline that we will not be discussing in this book.

Danger! Programming Project Manager

The Programming Project Manager is the equivalent of the player-coach in sports. It might work in amateur football, but it does not work in professional sports. And it does not work in enterprise application projects. If the project manager starts writing code as the deadline looms larger, project management deteriorates and the project ends up late and over budget.

The project manager should not be allowed to write code.

Software architect and lead programmer

The **Software architect** and the **Lead programmer** work together building an enterprise application like an architect and a builder work together to build a house.

The software architect designs the application, making key decisions about the use of the ADF framework, including the use of unbounded and bounded task flows, application module granularity and security. The lead programmer leads the team building the application. In principle, he could build the whole application himself, but in most real-life enterprise projects, he will build only a few key parts and spend most of his time supervising other programmers.

Both software architect and lead programmer need a good understanding of databases, internet technologies and the ADF Framework and must be willing to work with the grain of the tool.

Work with the Grain

Since ADF is a very powerful framework that does a lot of work for you, some developers distrust it and build a lot of code themselves. While this might be fun for the developer, it means that the business will be spending more time and money building a more fragile application. The lead developer must ensure that the ADF framework is used effectively and efficiently.

Apart from an enthusiasm for building enterprise applications fast with good tools, the Lead programmer, of course, also needs good Java programming skills and some years of experience.

Regular programmers

The regular programmers are the people doing most of the work. These are not necessarily great artisans—simply being a competent craftsman is enough to build great applications with ADF.

The programming tasks in your enterprise development project are likely to include:

- Building business components
- Building the user interface
- Skinning
- Templates
- Defining data validation
- Building support classes
- Building database stored procedures

In a typical project, the total functionality will be parceled out among team members, who will each be building both business components and user interface elements. However, large projects might specialize further with some people exclusively building business components and others building the user interface.

Other, more specialized development tasks will typically be handled by one or two persons in your team.

Building business components

While building business components, the developer will spend most of his or her time in JDeveloper, using the wizards and dialog boxes that are part of JDeveloper. During this task, the developer will occasionally need to read the XML files where JDeveloper stores the metadata for a business component, and will occasionally need to write short pieces of Java code when the default functionality of the business components do not meet the requirements.

Developing business components typically takes between one third and half of the total development time.

Building the user interface

Building the user interface is not a trivial task. In the JavaServer Faces technology used in ADF applications, components are not placed in specific locations, but are placed within layout containers. The layout containers determine how and where they are rendered on the screen, automatically resizing the components to make the best possible use of the browser window.

If you are used to tools with pixel precise placement of components, it will take some time to get used to working with layout containers. Programmers with experience building web applications using HTML tables will recognize some of the challenges you face, and programmers who have built Java swing applications will already be familiar with the concept of layout managers.

Much of this work can be greatly simplified by creating common design patterns and documenting their use on the project wiki or in a shared document.

The individual developer will normally be building page fragments and should focus on the functionality—the visual appearance is controlled through templates and skinning as described below.

The user interface also typically takes between one third and half the total development time.

Skinning

Skinning is normally understood as the process of removing the skin from an animal—to get to the meat, to use the fur, or both. However, the word also means, "to cover with skin". It is this second meaning that has been picked up and redefined in IT application development.

In ADF application development, skinning refers to the act of creating a **skin** for an application. The skin is purely visual and does not affect functionality—just like a panther (black leopard) is the same animal as a spotted leopard.

The person developing the skin needs a good understanding of Cascading Style Sheets (CSS). If your team includes people who have built web applications, they might already know some CSS. Developing an ADF skin can be a fairly difficult task; Oracle is working on an ADF Skin Editor to make this process simpler. Be sure to check the Oracle Technology website (`http://otn.oracle.com`, search for `adf skin editor`) to see if this editor is available by the time you read this book.

You do not need to define the visual attributes of every component—you start from one of the standard skins delivered with JDeveloper and change only those parts you want to look different. Always define a skin for your application when starting the project, even if you don't plan to change anything. Having a skin defined makes it easy to change the visual appearance of the application later.

You just need one person to develop and tweak the skin. This is a small, part-time task after the initial skin definition. You do not want every developer setting detailed styles on components— have the person doing the skinning explain the possibilities to the rest of the team to make maximum use of this powerful ADF feature.

Templates

In ADF, your pages are normally based on templates, and you can also define page flow templates that serve as the basis of your task flows.

These templates will often be built by the lead programmer or another ADF-experienced developer on the team.

Defining data validation

An enterprise application is likely to make extensive use of data validation to ensure that only the right data gets into database. You can define validation in all three layers of the application:

- In the user interface
- In the business components
- In the database

Validation in the user interface is handled by `Validate` operations you can place on user interface components. The advantage to this validation is that happens in the browser without placing any load on the server. A disadvantage is that a clever and malicious user will be able to circumvent this validation by disabling JavaScript in the browser or by modifying data before it is sent to your application—so you cannot depend on validation in the user interface alone.

A large number of validations can be defined declaratively in the business components; additionally, the ADF framework also allows you to specify validation using regular expressions, Groovy or Java code. These validation rules have the advantage that they are well integrated into the ADF framework and validation errors are easy to present to the user. A disadvantage is that business component validation does not happen until the user sends the data to the application server.

Finally, you can place validation rules in the database itself using PL/SQL code. These rules have the advantage that they will always be applied, no matter how the data gets into the database (through the application or any other way)—a disadvantage is that validation errors are more difficult to present to the user in the ADF pages.

Much of your validation will be built by the programmers building the business components and user interface; on a larger team you might want to appoint someone with experience of regular expressions as validation specialist.

Building support classes

Before the industrial age, a farmer would produce his own tools; using only the village blacksmith for the few tasks he could not do himself. As industrialization took hold, more and more specialized tools came into use—from steam-driven auto threshers to today's GPS-controlled combine harvesters. This has allowed a dramatic increase in output, but also means that the farmer is no longer able to repair his own tools.

Many developers tend to prefer the pre-industrial model when it comes to programming—they build their own utility classes and tools instead of leaving this task to a specialized tool builder. However, it is much more efficient to appoint someone from the team to build the utility and support classes you need. This ensures that each tool is only built once, is documented, and can be re-used through the whole application.

If your team is more than a handful of people, consider appointing one developer as the "tool builder," delivering the utility classes needed by other programmers. Because the code written by this person will be widely used in the application, the tool builder should be one of the more experienced team members.

Building database stored procedures

If you need to cut one log into planks, you can call a friend and get out your two-man saw. But if you need to cut a hundred logs, you ask a sawmill to do it.

If you need to process one data record, you can retrieve it from the database, process it in the application server and store it back in the database. But if you need to process a hundred, a thousand or a million records, you ask the database to do it.

If your application calls for this kind of batch processing, your team needs to include database programmers who can build stored procedures in your database. If your application is based on an Oracle database, the language of choice is PL/SQL—but other databases also have stored procedures in other languages.

Additionally, if data gets into your database through other means than the enterprise application, you need to implement validation and business rules at the database level to make sure that your logic is applied to all data.

Build/configuration manager

There are two parts to building an application: The thinking and the doing. The thinking is best done by a human, while the doing is best done by a machine. For more on the build machine, see the section "Automated Build System" later in this chapter.

The human is the build and configuration manager, who oversees the release and distribution of ADF libraries. When building an enterprise ADF application, your team will be working in different workspaces, and these workspaces will release their content in the form of ADF libraries. The build and configuration manager has the final word on which versions of libraries are used throughout the project and is the person who knows that version 0.3 of your application contains the common code library version 0.2.1 and version 0.4 of the CreateEmployee task flow.

This person is also in charge of the source control system, authorizing code branches and overseeing merges of code back into the main development track.

Try to avoid using a developer as build/configuration manager. A good build/configuration manager is careful, well-organized, and methodical. These are typically traits of a good systems administrator or database administrator, but not necessarily of a developer.

What is a good day?

When you are putting together the team for your enterprise application, ask each prospective participant this question: "What does a good day at work look like?"

The people who talk about new, exciting tasks should be developers. And the people who talk about all systems running smoothly should be build and configuration managers, database and system administrators.

Database and application server administrator

If your application was a ship, the database and application servers would be the engine – an indispensable part of the ship, but one that is rarely seem by most of the passengers and crew. And like the crew can use the engine order telegraph to send instructions to the engine room, an ADF developer can use the ADF Framework to send data and instructions to the database.

The ADF developer, of course, needs to understand the data model—just like the captain needs to understand the physical capabilities of his ship's engine.

Every captain knows that to keep a ship in operation, you need a trained and experienced professional in charge of the engine department: The chief engineer. An ADF application needs the equivalent: A database and application server administrator. A large organization might split this task into separate database administrator and application server administrator roles.

Apart from keeping the database and application servers humming along, the administrator can also help with performance problems. The ADF framework contains many, many ways of ensuring optimal performance, but it is still possible to run into a performance problem.

In some cases, this is caused by resource contention, for example, too many users trying to access the same application module at the same time. Your application server administrator can tell you in detail how your application is running and help you configure the many ADF tuning settings.

In other cases, this is caused by the application trying to tell the database *how* to answer a specific question instead of just asking the question and letting the database figure out the most efficient way of finding an answer. If you want the sum of all orders and your application is asking for each order separately, there is something wrong with your application. Just ask the database for the sum, and it will happily and quickly respond. Your database administrator can tell you what your application is really asking the database—if the database is not being asked the question you thought your application was asking, you need to look at your code.

Graphics designers

You need one or two graphics designers to create the visual identity of your application. Some organizations will already have a detailed style manual describing colors, fonts, and other visual attributes. In other cases, your designers have a free hand to create the look of the application.

Graphics designers tend to prefer working in very visual tools like Adobe Photoshop, sometimes leaving the programmers struggling with the challenge of converting their visual ideas into running code. To avoid wasting time designing something that cannot be built, ensure that your graphics designers have access to the ADF component catalog (Oracle is hosting an online demo at `http://jdevadf.oracle.com/adf-richclient-demo`).

If you do not have graphics designers in your organization, this task can be subcontracted. Graphics design is an area that lends itself well to online collaboration – you can send your specifications to a designer in another country, discuss drafts online and receive the finished design electronically. You can even hold online graphics design contents through sites like `www.99designs.com`.

Usability experts

To verify that your users can actually understand and use the pretty user interface your graphics designers have designed, you need usability experts to perform usability tests with prospective users. This must happen early in the development cycle when the screens might be little more than drawings on paper or simple wireframe mockups, and it is cheap and easy to change the UI.

If you can afford it, have at least two usability people on your team – two people can bounce ideas off each other and have a much better chance of coming up with creative ideas if you hit areas where your users just do not understand your application.

Because usability is a specialized topic, you can also consider subcontracting the whole usability task to an external supplier. However, this task cannot be sent out to someone to perform online – you need the usability people on-site with your users to capture all their input.

Cross-Training

While graphics design is a separate skill from usability, graphics design and usability people must work closely together – often, the same people will fill both roles.

Quality assurance, test manager, and tester

If you are working in an environment with strict regulatory requirements, or if you are a vendor delivering a software product to a customer for a fixed price, you are likely to need a Quality Assurance manager. This person has the responsibility to ensure that agreed quality procedures are followed.

Additionally, someone must be in charge of testing the application to prove that all requirements are met. For this, you need a Test Manager to write the test plan, ensuring full test coverage of the agreed functionality. The Test Manager also supervises the actual testing work and might be responsible for tracking all defects discovered by testing. Sometimes, the roles of QA Manager and Test Manager are combined in one person.

Finally, someone will have to write the detailed test scripts, possibly record automated test cases, and perform the test – this is the task of the professional testers.

Data modelers

If you are building your application from scratch or adding more than just a few tables to an existing data model, you need someone with an understanding of data modeling. The data model is the foundation of the whole application, and if the foundation it is not well built, the whole structure will be wobbly.

The inhabitants of Pisa in Italy might be happy to have a leaning tower today, but the duke commissioning the construction was probably not happy in the 12th century when his beautiful *campanile* started leaning.

Users

You are building the application because it will fill the needs of a specific group of users. You need user involvement for two purposes:

- To answer questions about the subject area
- For usability testing

For the first purpose, you need one expert that can explain the subject area and answer any questions you might have. If you are working with formal, detailed requirements documents, you might not be asking many questions, but if you are working according to an agile development methodology, it is imperative that an expert user is always available.

For the second purpose, you need multiple users with different experience levels to make sure you are building an application that is easy to use for the casual users and still efficient for the experienced power users. Depending on your ambitions for usability, you can involve anything from a handful of users up to several dozen.

Consider your representative users to be part of your team – send them the project newsletter, give them access to the project website, and so on. A new enterprise application will be a big change for many people; you want the users you work with to be your advocates and evangelists, spreading enthusiasm for the new application across the organization.

Worst practice: Design by Committee

Work with one person at a time – sit down with your expert to have your questions on the subject area answered, or present your user interface prototypes to one representative user at a time to gather feedback.

Having a committee of a dozen people or more questioning every aspect of the design and moving buttons and fields around on your screen designs will take months of extra development time without improving the final product.

Gathering the tools

In ages past, many people had such strong beliefs that they felt it was completely justified to burn people of different opinion at the stake. Mysteriously, similar strong passions often emerge among programmers when someone performs the heresy of proposing a different source control or build automation system.

The recommendations in this section are about the types of tools you need to manage your enterprise ADF project through the entire project development process and into production – commonly called **Application Lifecycle Management** (**ALM**) tools. Please do not burn this book if the recommendations do not match your particular programming religion – simply use the tool of your choice in each area.

Source control

In the 1930s, builders expected about one fatality per million dollars in construction costs. For the Golden Gate bridge in San Francisco, this meant that 35 people were expected to lose their lives. Actually, only eleven died. Nineteen people were saved by a revolutionary new invention: A safety net.

When building an enterprise application, your source control system is your safety net – the one tool you **absolutely**, **definitely** need.

Your source control system provides a centralized location for all your code – on a professionally managed server, securely located in your data center or with a hosting partner. This ensures that your precious source code is backed up, and that everyone can get access to all the code whenever they need to.

It also allows to you go back to an earlier version for whatever reason—the business might change their minds about the requirements (yes, it has been known to happen), you might implement a change which proves to have unforeseen side effects, a file might become corrupted due to a hardware failure. There is no excuse for not having source control in place before development starts.

A good source control system is Apache Subversion (SVN). It is fairly widespread, so your development team is likely to have encountered it before. It integrates directly into JDeveloper, and there are many stand-alone clients available.

So before you write the first line of code of your real application, talk to your systems management people to find out what source control system your organization is already using. If there is no system available for your development project, you can easily set up a source control repository on a development server. With the low cost of hard drives today, it is recommended to configure this server with multiple disks in a fail-safe RAID configuration to avoid disrupting development in case of a hard disk failure. Additionally, talk to your system administrator to make sure the machine with your source code repository is included in the daily backup routine.

Bug/issue tracking

"In the unlikely event" (as they say in the airline industry) that your code does not work perfectly the first time, you need some system to track issues and defects. Many of these tools are also used to assign tasks to developers.

There is a wide variety of tools available, for example:

- Jira: A popular, commercial tool. Full-featured and user-friendly. Mobile clients available (just in case you want to read bug reports on your iPhone). Free for non-commercial use.
  `www.jira.com`
- Bugzilla: An open source bug-tracking tool that scales to many thousands of bugs/issues (not that you'd have that many, of course). More focus on functionality than user interface – can be fairly intimidating for new users.
  `www.bugzilla.org`
- FogBugz: Another commercial tool for complete project management, including bug and issue tracking. Focus on support to end users as well with automated classification of email bug reports and a screenshot tool for testers and end users.
  `www.fogbugz.com`

All of these can either be hosted externally or installed locally – again, talk to your systems management department to see if your organization already has a bug/issue tracking system that you can use. If not, you can install one of these tools on a development server.

If you use a project development server for bug/issue tracking, make sure that it is backed up regularly. For an ADF project, Jira and Bugzilla have the advantage that Oracle has already built adapters integrating them into Oracle Team Productivity Center (OTPC). However, if you already have a bug/issue tracking system running, there is no need to throw your existing system away. It is perfectly feasible to run your bug/issue tracking system without integrating it with OTPC—or you could build an OTPC adapter yourself.

Collaboration

Even if every member of your team works in the same office and you have regular project meetings, you still need collaboration tools to facilitate communication in the team. Several types of collaboration tools can be useful:

- Shared documents
- Discussion forums
- Online chat

There are many free, open source tools available for all of this. If you would rather have something pre-built and pre-integrated, you can subscribe to web-based services or even buy sophisticated (and pricey) tools like Oracle WebCenter Spaces (part of the Oracle WebCenter Suite).

Shared documents

The most basic collaboration tool is shared documents, and you definitely need this. This is where you put your development guidelines, and naming and code standards. At the simplest, this can be a wiki or shared online documents like Google Docs.

The important part about shared documents is to make sure that everyone can edit them. You do not want to have to wait for the project manager to come back from some meeting before you can update a standards document with some great idea or much-needed improvement.

Because everyone can edit the documents, the other feature you need is a full version history so that you can easily go back to a previous version and compare versions.

Discussion forums

Stop sending e-mails to your team. Just stop it. E-mail is so last century.…

Seriously, e-mail is not really useful for collaboration.

For one thing, everybody's email inbox is already overflowing with meeting invitations, promotional e-mails from all the companies that you gave your business card to at the last conference in return for a free T-shirt, funny pictures of cats your friends think you need to see, Viagra ads, and mails from friendly people who need to borrow your bank account to transfer a couple of million dollars. This almost ensures that someone will miss the crucial e-mail you send about changes to how code should be checked into source control.

Secondly, collaboration is not about individual messages, but about discussions. And e-mail programs do not really show discussions well. What you need is a real tree view of who said what and in which order. If someone changes the subject of the e-mail slightly, the e-mail program gets confused and is unable to follow the thread.

So do set up a discussion forum – if you do not have the software, you can use a private group on Google Groups or another online service. One-to-many communication and announcements go into the shared documents, and all other communication goes to the discussion forum.

Online chat

Some people love online chat systems, and other people cannot be bothered. This is not a required part of your collaboration toolkit, but if you think it might be useful, try it out.

There seems to be a generational divide in online chat – younger people who have grown up with multiple forms of digital communication tend to like it, and older people tend to find that constant online chatter impedes their productivity.

The teams that typically achieve the greatest benefit from using an online chat application are geographically dispersed teams in approximately the same time zone. If you are all in the same office, you can get help by walking down the corridor, and if you are in different time zones, the person with the answer might not be online when you need to ask a question.

Oracle Team Productivity Center, which we will talk about in the next chapter, contains a chat client.

Test and requirement management

Your quality assurance manager and test manager can work with spreadsheets and documents, carefully mapping test cases to requirements. However, for an enterprise project, you should look into professional software solutions (like HP Quality Center and similar) for managing requirements and test cases. Many of these professional tools will also allow you to automate your application testing.

Automated build system

The final tool you should strongly consider is a build tool. Think about all the tasks involved in building your application:

- Check out the latest version of the code from your source control system (you do have one, don't you?)
- Compile all code
- Run all the unit tests you have defined
- Generate Javadoc and/or other documentation
- Audit the code for quality issues
- Package the whole application into a deployment package
- Deploy it on your test/integration server
- Run any automated user interface tests

Once you have set up your build tool, you can have the tool perform all of these tasks automatically – saving developer time, reducing bugs due to manual errors and improving the quality of your code.

Several tools are available for this, for example:

- Apache Ant (`http://ant.apache.org`)
- Apache Maven (`http://maven.apache.org`)

In addition to the pure build tools that actually execute the tasks you define, you can also use **continuous integration** tools that will automatically invoke your build tool based on specific rules. For example, you might compile and run unit tests every time a developer checks something in to your source code repository, and run a complete build every night at 2:00 a.m. Tools for continuous integration include:

- Hudson (`http://hudson-ci.org`)
- CruiseControl (`http://cruisecontrol.sourceforge.net`)

All of the above tools are open source, but there are commercial tools available as well. If your organization or someone on your team already has experience with one of them, stay with that tool. If you do not have any preferences, have someone spend a day researching tools and choose the one the appeals the most to you.

Structuring workspaces, projects, and code

Your ADF application is going to consist of lots of files and development artifacts. To prevent utter chaos, you need to decide how to divide your enterprise project into manageable parts. Fortunately, JDeveloper and the ADF framework support this separation very well through the concepts of **workspaces** and **ADF Libraries**.

Workspaces

JDeveloper works with the concept of application workspaces (confusingly often just called "applications"). To avoid this confusion, the following will refer to application workspaces as just **workspaces**. Each application workspace can contain multiple projects.

You should *not* build your entire enterprise application in just one application workspace. Having everything in one big workspace places an unnecessarily heavy load on JDeveloper that will be noticeable, even on a powerful development workstation. Additionally, because you work with version control at the workspace level, having one big workspace means that your source control operations (updates and commits) will take longer.

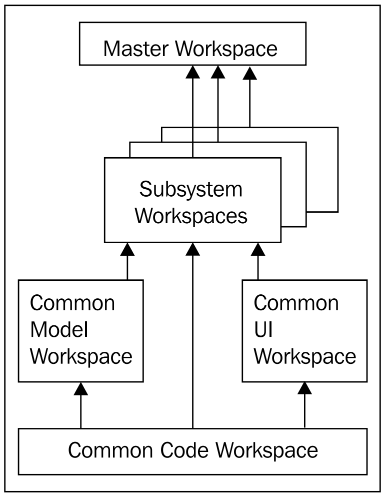
Show me only what I need

If you find that JDeveloper is showing you too many objects that are not relevant to your task, you can define **Working Sets**. These are configurable filters on your application workspace that you can define under **Application | Filter Application**.

Instead of one big workspace, partition your application as follows:

- One common code workspace
- One common UI workspace
- One common model workspace
- One database workspace (unless the database already exists)
- A number of subsystem workspaces
- One master workspace

Each workspace should be deployed as an ADF library that is placed under version control. Workspaces that depend on objects from other workspaces will then import the latest ADF Library released by that workspace. The subsystem workspaces will include the ADF libraries released by the three common workspaces, and the master workspace will include all the subsystem workspaces.



Get it right the first time

Think carefully about how you want to split your system into subsystems. A subsystem should represent a logical grouping of functionality to minimize dependencies between subsystem workspaces, and should be implemented by a small team of no more than four developers.

Unfortunately, it is not easy to move objects and code between workspaces once you have started building. Only change the division of code in subsystems if you find major issues.

Common code workspace

The common code workspace is where you place your framework extension classes and any utility classes you develop.

The people who work here will be the hardcore Java and ADF coders—the lead programmer or the most senior programmers on the team. These classes should be well written and extend existing classes in the right way. The work done in this workspace will affect the entire project, so it is important to get this code right.

Once the first version of the common code ADF library has been released, new releases should be rare and will necessitate serious regression testing across the entire project.

Common user interface workspace

The Common UI workspace is where you keep all the common elements that define the visual identify of your application. This includes skins, page templates, and page flow templates.

Both the skin developer and the page and page flow template developers work in this workspace. Unless you have a large team, these two roles are likely to be filled by the same person.

A skin defines the look of an application. Even if you do not anticipate customizing the look of the application, you should create your own skin at the beginning of the project. It does not take much effort to create a skin (you can base it on one of the skins delivered with JDeveloper), and it gives you a place to change the visual identity of the application if you later decide you need this. We will return to skinning in *Chapter 8, Look and Feel*.

Expect many minor changes to happen in the user interface template workspace, mainly to the skin or visual identity of the application. Changes that only affect the skin can be rolled out without affecting the functionality of the application. Changes that affect the page and page flow templates need a bit of regression testing to ensure that pages still work as you expect them to.

Common model workspace

In the common model workspace, you keep all of your entity objects for the whole application. Since there will be only one entity object for each relational table, it makes sense to gather these together in one workspace and create an ADF library that can be shared with all dependent projects.

This workspace will also contain the view objects built for use in value lists across the application and the application module containing these view objects. Additionally, this workspace needs default view objects for each entity object so you can test the EOs.

Once the initial components in this workspace have been built, only the validation programmer is likely to be working much in this workspace, with occasional visits from a business component developer if the need for a new value list pops up.

Expect one initial release containing all the entity objects, and then a number of releases with small changes involving validation rules as the project progresses. Unless you make major changes involving the removal of entity objects, you do not need much regression testing when releasing a new version of the ADF library from the common model workspace.

Database workspace

If you keep your data model in JDeveloper, you should have a separate workspace for the offline tables and other data elements, as well as any database diagrams you use.

You do not need to keep your database definition in JDeveloper. While it might be easier to use the same tool for everything, there is no real integration benefit from having your data model in the same tool as your business objects. If you prefer to use Oracle SQL Developer Data Modeler or another tool you are already familiar with, feel free to do so.

Subsystem workspaces

You will generally be implementing one task flow for each use case or user story. This means that your application can have anything from a handful to hundreds of task flows.

You should group these task flows together in subsystem workspaces, each developed by a small team. That team will be creating the specific view objects necessary for the task flows in the model project of the workspace, and the task flow and pages in the view/controller project of the workspace.

A subsystem workspace will import the common workspaces through the latest ADF Libraries released by each of these sub-projects.

Master workspace

The master workspace is where the build and configuration manager puts everything together. This workspace depends on all the other workspaces and contains no code of its own—it simply serves as a container for all of the task flows your application consists of. The final application package is built from this workspace.

Using projects

Within an application workspace, you can have one or more projects. Your common workspaces will typically contain only one project, but your subsystem workspaces will always contain at least two or three projects:

- Model project
- View/controller project
- Test project

When you create a new application workspace of type **Fusion Web Application**, JDeveloper will automatically create a model and a view/controller project for you. You should not use the default project names, as this will cause problems when you combine ADF libraries from multiple projects in the master workspace.

Instead, name both projects after your subsystem with the suffixes `Model` and `View`, respectively. For example, if your subsystem is called `TaskHandling`, you should use project names `TaskHandlingModel` and `TaskHandlingView`. You need to create the test project after creating the application workspace, using a name like `TaskHandlingTest`.

Naming conventions

The better your standards – and the better they are adhered to – the easier your code will be to develop and maintain. You can use some or all of the following standards and add your own. Whenever JDeveloper automatically generate names for things, consider if these are good enough—only write a naming standard that deviates from the standard JDeveloper way if you feel the rule has significant benefit.

Your standards should be easily available to everyone on the project in a wiki or a shared document.

General

You need to decide on an application name: An abbreviated project name or an acronym that can be used in package names. You are going to be writing this over and over, so make it as short as possible – preferably just 3 or 4 characters, for example, `xdm` (for project "neXt generation Destination Management").

Java packages

All your business components and your Java code will be placed in packages and displayed in the package hierarchy in the Application Navigator in JDeveloper. To make it easy for all team members to find everything, determine your Java package naming hierarchy from the beginning.

As you remember from *Chapter 1*, *The ADF Proof of Concept*, the base of all your Java package names is by tradition the internet domain name of your organization with the elements in reverse order. Under this base package, all code belonging to the project should be placed under your project name – for the XDM project of DMC Solutions, this would be `com.dmcsol.xdm`. This is called the **project base package**. All of your code should be placed in sub-packages under the project base package.

You package naming could look like the following table (where <*pbp*> is short for the project base package):

Package	Usage
<*pbp*>	Project base package, no code directly here
<*pbp*>.framework	Framework extension classes
<*pbp*>.common	Common utility classes
<*pbp*>.model.entity	Entity objects (always common to whole application)
<*pbp*>.model.entity.assoc	Associations between entity objects. Also common to whole application
<*pbp*>.model.view	Common view objects (for example, for lists of values used across the whole application)
<*pbp*>.model.resources	Resource bundles for the common model
<*pbp*>.<*subsystem*>.model.view	All view objects built specifically for <*subsystem*>
<*pbp*>.<*subsystem*>.model.view.link	View links for <*subsystem*>
<*pbp*>.<*subsystem*>.model.am	Application module(s) for <*subsystem*>
<*pbp*>.<*subsystem*>.model.resources	Resource bundles for the <*subsystem*> view objects
<*pbp*>.<*subsystem*>.view	Java classes common to the frontend part of the application (for example, managed beans for the task flows of the subsystem)
<*pbp*>.<*subsystem*>.view.backing	Backing beans for individual pages
<*pbp*>.<*subsystem*>.view.resources	Resource bundles for the frontend part of the application (page flows, pages and page fragments)

Database objects

If you are working with an existing database, you should use the existing naming conventions. If these are not documented, look at existing objects and use similar naming – do not change the naming if you inherit an existing database, even if the naming conventions are not to your liking.

If you are building a new database, you need to set naming standards at least for the following database objects:

- Tables
- Columns
- Views
- Primary key constraints
- Foreign key constraints
- Sequences
- PL/SQL packages
- Triggers

Table names should be prefixed with the application abbreviation, even if they reside in a database schema only used for the application. During the lifetime of your enterprise application, people are likely to be using the data in your tables together with other data for purposes nobody has thought of from the beginning. Prefixing your tables names clearly identifies them as part of a specific application. Each table name should be the plural of the record it contains, for example, `XDM_TASKS`.

Primary key **columns** should be named with the table abbreviation (3-8 characters) and the suffix `_ID`, if they contain a system-generated number. Use the suffix `_KEY` if the value is an alphanumeric key. For example, `PROG_ID` or `ELEM_KEY`.

Normal **views** should have the suffix `_V`. If you are using updatable views with INSTEAD-OF triggers, use the suffix `_UV`.

Primary key constraints should have the name of the table with the suffix `_PK`, for example `PERSONS_PK`.

Foreign key constraints should be named with the abbreviation of the table where the constraint is defined, followed by the abbreviation of the table referred to, with the suffix `_FK`. For example, `TASK_ELEM_FK`.

Sequences should have the suffix `_SEQ`.

Triggers should have the name of the table with a three-part suffix indicating their usage:

- The first part should be `B` or `A`, indicating `BEFORE` or `AFTER`
- The second part should be a combination of the letters `I`, `U` and `D`, indicating `INSERT`, `UPDATE` or `DELETE`
- The third part should be `R` or `S`, indicating `ROW` or `STATEMENT` level

An example of a trigger name could be `TASKS_BIR`: Before insert at the row level.

ADF elements

You should set naming standards for the following ADF elements:

- Entity objects
- Associations
- View objects
- View links
- Application modules
- Task flows
- Pages

Entity objects should have the same name as the table they are based on, but singular and without the project prefix. The entity object for the `XDM_TASKS` table should thus be called `Task`.

By default, **associations** get the name of the foreign key they are based on, with the suffix Assoc. There is no need to change this.

View objects should be named for functionality or data they include – there is no need to force an artificial correspondence to entity object names. Use a `VO` suffix for the main view objects, for example, `EmploymentHistoryVO`. Use a `LOV` suffix for simple view objects that are only used for value lists, for example, `ServiceLOV`.

View links should be named with the detail view objects (without the VO suffix), followed by the master view object (without the VO suffix) and the suffix `Link`. For example `TaskPersonLink`.

Application modules should be named according to the service they provide (typically, the name can be derived from the subsystem that uses the application module). Use the suffix `Service`. Inside application modules, JDeveloper by default gives the individual view objects usages the name of the view object with a numeric suffix (for example, `TasksVO1`). In most cases, the numeric suffix is unnecessary and should be removed when adding a view object instance to an application module.

Task flows have both a **Task Flow ID** (which is also the name of the XML file) and a **Display Name** that is shown in the Application Navigator in JDeveloper. The ID should be brief and might refer to the use case number or user story it implements. Use the suffix `-flow`, for example `timeline-flow`. Use **Display Name** if you feel you need a more detailed title to identify the task flow.

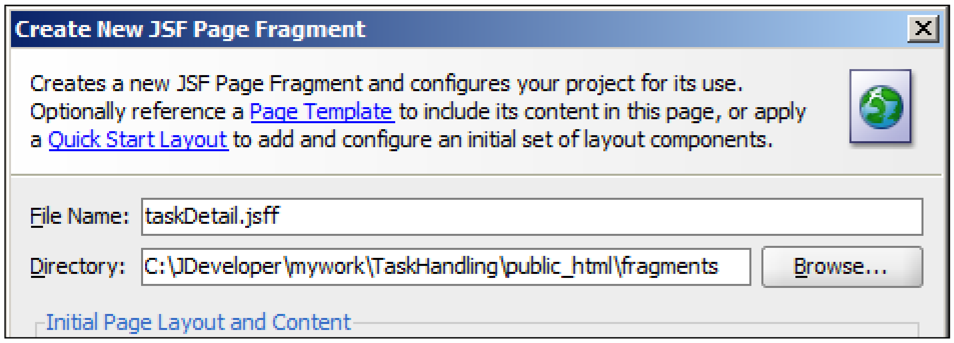
File locations

Each project has a base directory in the file system that is defined when the project is created. Under the base directory, a `src` subdirectory is created, and all your Java code and ADF business component XML will be placed in subdirectories under `src` matching your package names.

Your ADF view/controller project also contains a subdirectory called `public_html`. The content of this directory is shown in JDeveloper under the **Web Content** node in the Application Navigator. Everything in this directory is accessible from a browser when the application is running—except for the content of the subdirectory `WEB-INF`.

Pages should be left in their default location (**Web Content** in JDeveloper, corresponding to `public_html` in the file system).

Page fragments should be placed in a subdirectory `fragments` to separate them from the pages. When creating page fragments by double-clicking on a view object from a task flow, you can simply add this directory name to the end of the content of the **Directory** field. This causes JDeveloper to create the subdirectory:



Creates a new JSF Page Fragment and configures your project for its use. Optionally reference a Page Template to include its content in this page, or apply a Quick Start Layout to add and configure an initial set of layout components.

File Name: taskDetail.jsff

Directory: C:\JDeveloper\mywork\TaskHandling\public_html\fragments

Your **page flows** and **page templates** do not need to be directly accessible, so you should place these under the `WEB-INF` directory. This prevents a malicious user from snooping around in the application's internal files.

Test code

If you are using the Maven tool, you should follow the Maven conventions for the placement of test code. If not, create a separate test project in each application workspace and place your unit tests in this project.

Having the tests separate from the code makes it easier to deploy the final application without including test code unnecessary at runtime.

Summary

You have learned which skills you need in order to build an enterprise ADF application. Talking to your colleagues in the development department of DMC Solutions, you found that most of the skills were available, but you also had to find external resources to fill a few of the roles in the development team for the XDM project.

Talking to the system administrators in DMC Solutions, you found that there was already a Subversion repository available you could use for your project, but you had to install a couple of additional tools for issue tracking and collaboration on your development server.

On the project Wiki, you have described how your team should use application workspaces to build the application in a modular fashion, and you have documented the naming conventions that everyone on the project should use.

The next task is to get the development team together for the official kick-off and tell everybody how to use the tools, so you can build the XDM solution in the most efficient way. That is the topic of the next chapter.

4
Productive Teamwork

You have put together your team so you that all the necessary skills are available and you have set up your development servers (versioning, issue tracking and collaboration). You have already documented the XDM project standards in the project Wiki - now you need to get everybody together for the kick-off and the first official project session, telling everybody how to set up their development workstations to work productively with the tools.

The secret of productivity

Don't you just hate it when you're in the middle of something and your boss comes up to you with an urgent request that your just *have* to start on right away? You need to put away what you are doing, get into the new task, finish it and then get back into what you were doing before.

Productivity experts call this a **context switch**. And it is much more expensive in terms of lost productivity than most people realize. Estimates of lost time vary, but typically range from 10 minutes to an hour of lost productivity. To be more productive, you need fewer context switches.

This book cannot really offer you advice on how to minimize boss-induced context switches – but it can offer some guidelines for minimizing the context switches you can influence yourself.

More pixels, better productivity

When you start working with JDeveloper, you are likely to find that your screen is too small. If you are wasting time minimizing and resizing panels, you need a bigger screen. Get at least a 1600 x 1200, preferably bigger (or even better, get two). With current monitor prices, the payback time for extra screen area can be calculated in weeks.

Integrate your tools

An experienced computer user can switch applications in a fraction of a second – but the context switch takes much longer than the time it takes to press *Alt+Tab* or *Cmd+Tab*.

Firstly, you are likely to have more than two applications you switch between – which means that every once in a while, you will have to actually look at your screen to determine just how many *Alt+Tab* presses you need. Secondly, once the desired application is on the screen, you need to reorient yourself *every time*. Your brain has to determine the location of the window, find the mouse pointer and then instruct your hand to scroll or move. Thirdly, you will occasionally have to actually use information from the second window in the first one. You can either resize both windows to a smaller size or cut and paste the information you need – both of which take time.

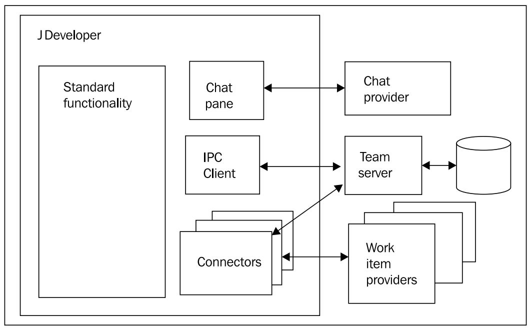
That is why you do not want to have to switch applications – you want as much as possible integrated into your development environment.

The Oracle solution

In days past, Oracle's solution to this integration challenge would have been to connect JDeveloper to some of their own software, like the Oracle Projects module of the E-Business Suite. Fortunately, Oracle has moved on from the monolithic "buy-everything-from-us" approach and with Oracle Team Productivity Center (OTPC) offers a modern, loosely coupled integration solution.

Oracle Team Productivity Center is a free product from Oracle that connects to different repositories of information and presents data directly in JDeveloper. The Team Productivity Center consists of three parts:

- JDeveloper TPC Client
- Team Productivity Center Server
- Team Productivity Center Connectors



The JDeveloper TPC Client is implemented as a JDeveloper extension. This means that you can simply use the built-in **Check for Updates** function to automatically download and install the client.

The Team Productivity Center Server is a Java application offering a number of services that the client uses. It contains a password vault storing credentials for third-party repositories so the user can seamlessly connect to different products, as well as a simple task database in case you do not have any third-party repositories you want to use.

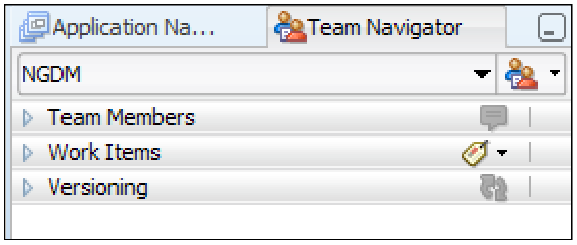
The Team Productivity Center Connectors expose third-party repositories of information to the client. Using a standards-based interface, a connector allows create/update/delete operations on the underlying repository, so the developer can do all work without leaving JDeveloper. Team Productivity Center comes with pre-built connectors for Jira, Bugzilla, Rally, and Microsoft Project Server, as well as the built-in simple task repository.

If you want to integrate a tool for which Oracle does not yet offer a connector, you can develop your own connector. Oracle makes a Developer Guide available explaining how to develop a connector, as well as a sample connector with source code to help you get started.

Team Navigator

When you install Oracle Team Productivity Center, you get a **Team Navigator** entry under **View | Teams**. If you choose this menu item, the Team Navigator pane opens. This pane shows three expandable headings:

- **Team Members**
- **Work Items**
- **Versioning**



Under **Team Members**, you find all members of your team. If you have connected to a chat server as described below, you also see the status of each member (available, away, busy).

Under **Work Items**, you see the **Active Work Item** at the top, followed by all the repositories you have connected to (Jira, Rally, Bugzilla, and so on.). Under each repository, lists of work items from that repository are shown. Oracle Team Productivity Center does not store the work items itself—you define **queries** either at the team level or just for yourself, and these queries show a number of work items.

Under **Versioning**, you find connections to the versioning system you use on the project. The functionality here is the same as in the standalone **Versioning Navigator**; the advantage to defining versioning as part of Oracle Team Productivity Center is that the team administrator can specify the repository URL for the team.

Chat

Also on the **View | Team** menu, you find a **Chat** item that brings up a **Chat** pane inside JDeveloper.

The first time you click on the connect icon, you will be asked to define a connection to a Chat server. You can connect with any XMPP/Jabber talk client, for example, Google Talk.

Once you are connected, you see your team members as well as buddies from this chat server with an icon showing status, just like in other chat clients.

Oracle Team Productivity Center

If you decide to use the recommended Oracle productivity solution, you need to install the Oracle Team Productivity Center server on a server in your environment. If you do not want to allocate a dedicated server for this (you do not have to), you can install the TPC server components on your test/integration server.

Installing the server

The Oracle Team Productivity Center server needs a database with a schema where it can store its items, and a Java application server to run the server-side code. The database can be any database that allows a JDBC connection – for example, MySQL, the free Oracle XE database or one of the commercial Oracle database versions. If you decide to install the Oracle Team Productivity Center server components on your test/integration server, you can use the database already in place – simply create a separate schema for the OTPC components. The application server can also be any JEE application server – if you install on your test/integration server, you can use the same WebLogic server you use for your test. If you install on a separate server, a free, open source product like Tomcat will be sufficient.

You can find the installer on the Oracle Team Productivity Center web page on the Oracle Technology Network website (`otn.oracle.com`) – at the time of writing, it could be found at `http://otn.oracle.com/developer-tools/tpc/downloads`. Use the site search feature to search for `Team Productivity Center` if you cannot find it

On the Oracle Team Productivity Center page, accept the OTN license and download the Team Productivity Center Server and any connectors you need.

Once you have downloaded the install file `tpcinstaller.jar` to the server where you want to install it, open a command prompt. Then set JAVA_HOME to point to your JDK directory and set the PATH to include JAVA_HOME/bin. If your OTPC server is running Microsoft Windows, the commands could look like this:

```
C:\install>set JAVA_HOME=c:\java\jdk1.6.0_22
C:\install>set PATH =c:\java\jdk1.6.0_22\bin;%PATH%
```

Then run the installer with the following command:

```
C:\install>java –jar tpcinstaller.jar
```

The Oracle Team Productivity Center installer starts.

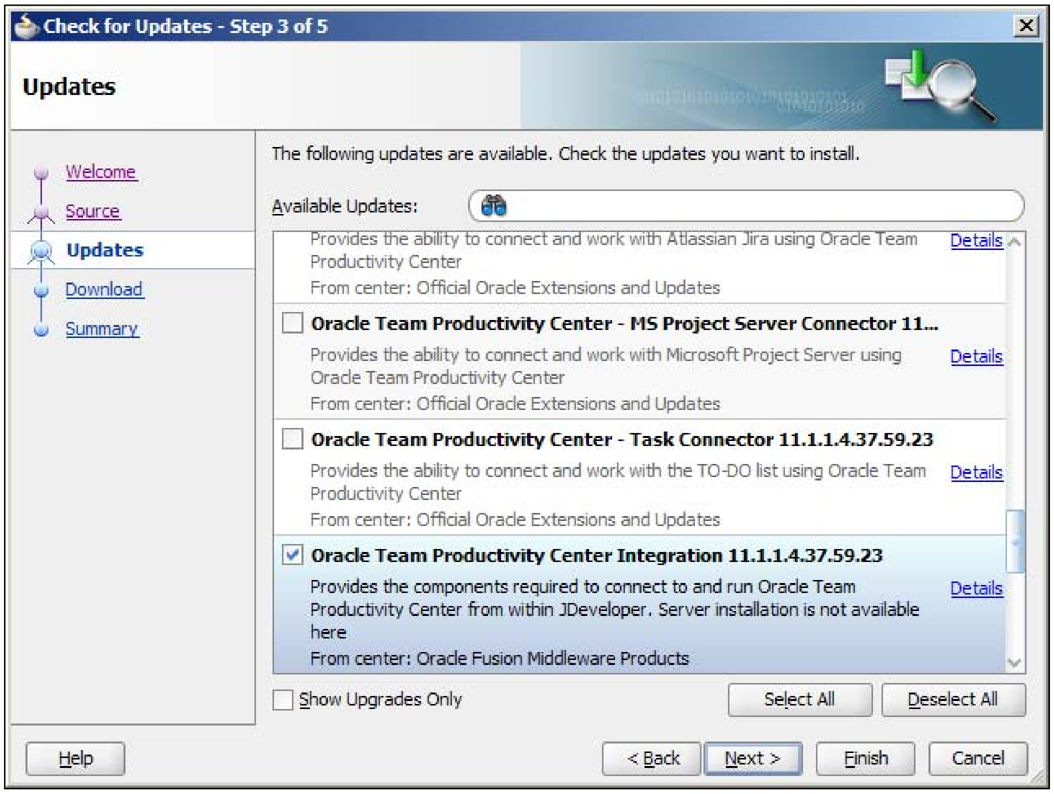
In Step 2, provide connect information to the database schema where you want OTPC to store its information. This must be an existing schema with `CONNECT` and `RESOURCE` privileges (or similar, if you are not using an Oracle database).

In Step 5, where you are asked for an application server location, you need to point to an auto-deploy directory – a directory where the application server will automatically detect new applications. If you are installing Oracle Team Productivity Center in Oracle WebLogic, the name of your directory is of the form `<weblogic_home>/user_projects/domains/<your_domain>/autodeploy`, for example `c:\oracle\Middleware\user_projects\domains\xdm_test\autodeploy`. If you are using Apache Tomcat, this is the `/webapps` directory under your Tomcat base directory.

In Step 6, you can choose the local connector `.zip` file if you have downloaded one from the OTN website, or you can download connectors directly from the Oracle Update center.

Installing the client

It should be possible to install the Oracle Team Productivity Center client in JDeveloper through the automatic update feature (**Help | Check for Updates**). In the list of update sources, leave **Oracle Fusion Middleware Products** and **Official Oracle Extensions and Updates** checked and click **Next**. In the dialog showing available updates, just write **team** in the search field at the top to limit the list as shown in the illustration next:



The following updates are available. Check the updates you want to install.

Available Updates:

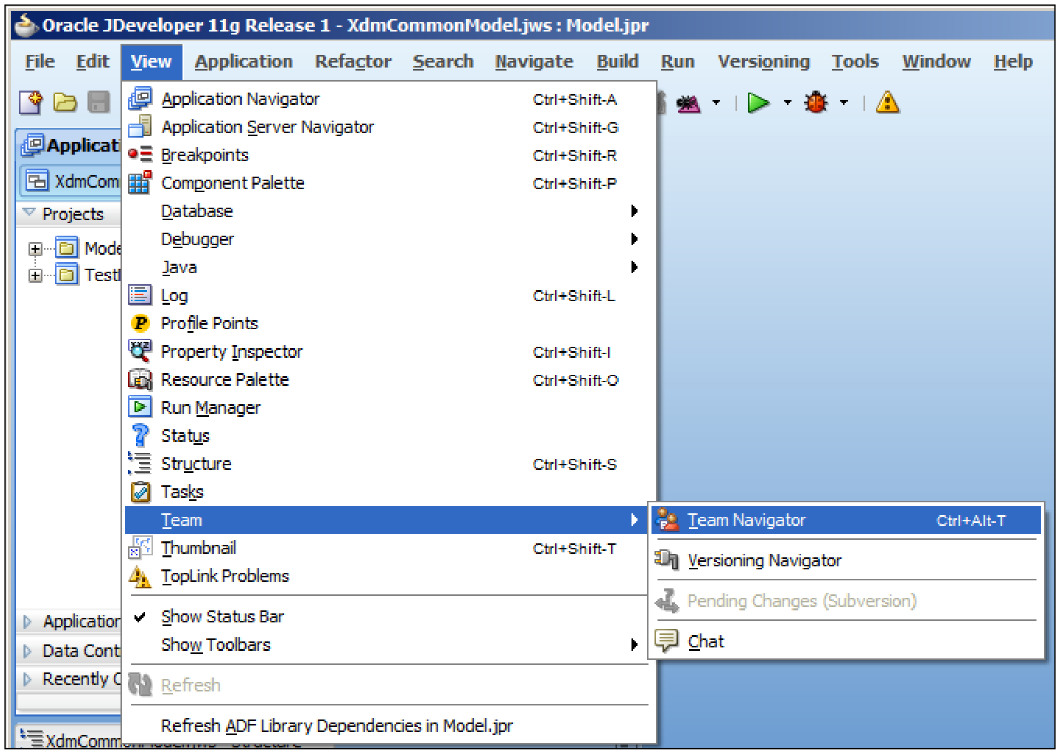
Provides the ability to connect and work with Atlassian Jira using Oracle Team Productivity Center
From center: Official Oracle Extensions and Updates

☐ **Oracle Team Productivity Center - MS Project Server Connector 11...**
Provides the ability to connect and work with Microsoft Project Server using Oracle Team Productivity Center
From center: Official Oracle Extensions and Updates

☐ **Oracle Team Productivity Center - Task Connector 11.1.1.4.37.59.23**
Provides the ability to connect and work with the TO-DO list using Oracle Team Productivity Center
From center: Official Oracle Extensions and Updates

☑ **Oracle Team Productivity Center Integration 11.1.1.4.37.59.23**
Provides the components required to connect to and run Oracle Team Productivity Center from within JDeveloper. Server installation is not available here
From center: Oracle Fusion Middleware Products

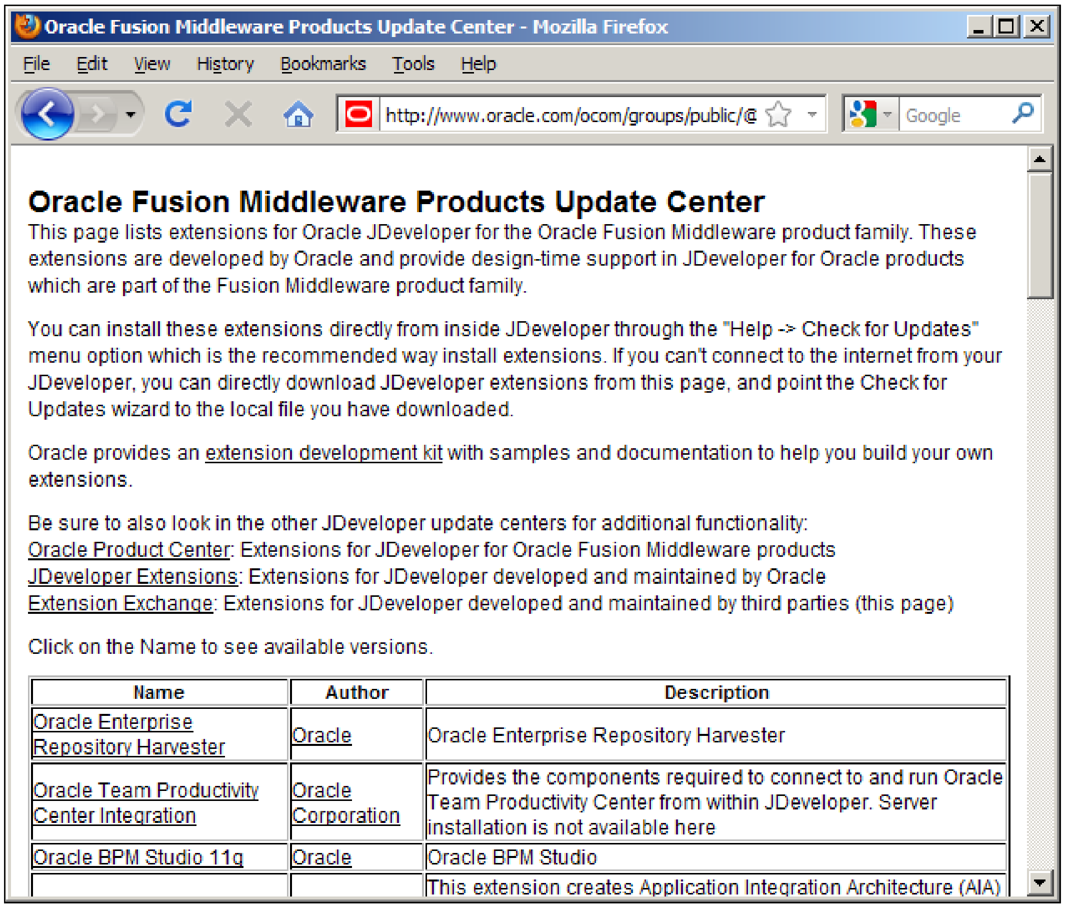
Select the Oracle Team Productivity Center client itself as shown previously, and any connections your need. When you click **Next**, the extension is downloaded and installed. Then click **Finish** and allow JDeveloper to restart. When you select **View | Team**, you should see two new sub-menus **Team Navigator** and **Chat** as shown next:


Oracle JDeveloper 11g Release 1 - XdmCommonModel.jws : Model.jpr
File Edit View Application Refactor Search Navigate Build Run Versioning Tools Window Help
Application Navigator Ctrl+Shift-A
Application Server Navigator Ctrl+Shift-G
Breakpoints Ctrl+Shift-R
Component Palette Ctrl+Shift-P
Database
Debugger
Java
Log Ctrl+Shift-L
Profile Points
Property Inspector Ctrl+Shift-I
Resource Palette Ctrl+Shift-O
Run Manager
Status
Structure Ctrl+Shift-S
Tasks
Team
Thumbnail Ctrl+Shift-T
TopLink Problems
Show Status Bar
Show Toolbars
Refresh
Refresh ADF Library Dependencies in Model.jpr
Team Navigator Ctrl+Alt-T
Versioning Navigator
Pending Changes (Subversion)
Chat

If you do not see these new menu items, it is likely that the automatic install failed for some reason. To manually install an extension, do the following:

1. Choose **Help | Check for Updates** to open the Check for Updates dialog.
2. Make the dialog wider so you can see the URL of the Oracle Fusion Middleware Products update center (something like `http://www.oracle.com/ocom/groups/public/@otn/documents/webcontent/156082.xml`). Make a note of the URL and open this in a Web Browser.

3. You will see the update center web page as shown next:


Oracle Fusion Middleware Products Update Center - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.oracle.com/ocom/groups/public/@
Google

Oracle Fusion Middleware Products Update Center

This page lists extensions for Oracle JDeveloper for the Oracle Fusion Middleware product family. These extensions are developed by Oracle and provide design-time support in JDeveloper for Oracle products which are part of the Fusion Middleware product family.

You can install these extensions directly from inside JDeveloper through the "Help -> Check for Updates" menu option which is the recommended way install extensions. If you can't connect to the internet from your JDeveloper, you can directly download JDeveloper extensions from this page, and point the Check for Updates wizard to the local file you have downloaded.

Oracle provides an extension development kit with samples and documentation to help you build your own extensions.

Be sure to also look in the other JDeveloper update centers for additional functionality:
Oracle Product Center: Extensions for JDeveloper for Oracle Fusion Middleware products
JDeveloper Extensions: Extensions for JDeveloper developed and maintained by Oracle
Extension Exchange: Extensions for JDeveloper developed and maintained by third parties (this page)

Click on the Name to see available versions.

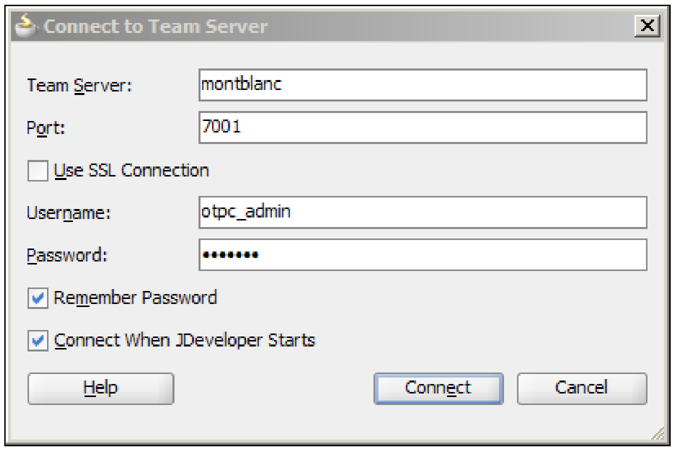
Name	Author	Description
Oracle Enterprise Repository Harvester	Oracle	Oracle Enterprise Repository Harvester
Oracle Team Productivity Center Integration	Oracle Corporation	Provides the components required to connect to and run Oracle Team Productivity Center from within JDeveloper. Server installation is not available here
Oracle BPM Studio 11g	Oracle	Oracle BPM Studio
		This extension creates Application Integration Architecture (AIA)

4. Scroll down on the page to the Oracle Team Productivity Center section and click on the **Download** link next to the highest version number.
5. Go back to the **Check for Updates** dialog and choose **Install from Local File** at the bottom of the dialog and browse to the `tpc_bundle.zip` file you downloaded. Click **Next** and then **Finish** and allow JDeveloper to restart.
6. Repeat this procedure for any connections you need. Note that the connectors are found in the **Official Oracle Extensions and Updates** section, which has a different URL.
7. When done, restart JDeveloper.

Administration tasks

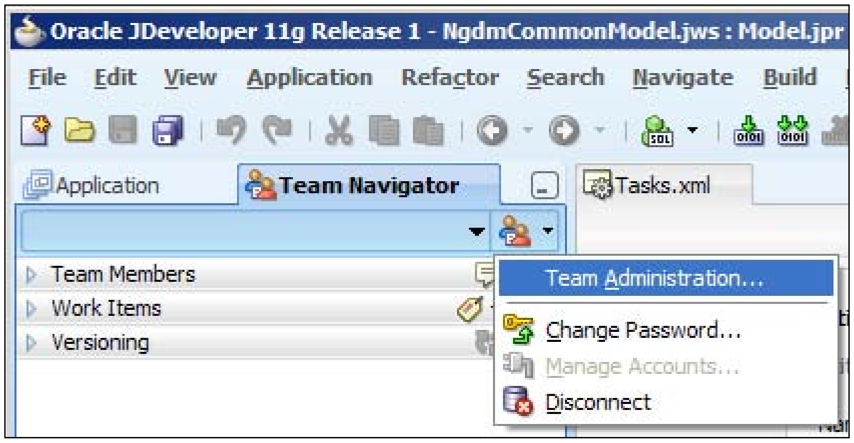
When you have both the client and the server installed, choose **View | Team | Team Navigator** to bring up the **Team Navigator** pane. By default, it shows up in the top left part of the JDeveloper window, together with the **Application Navigator**, **Run Manager** and **Connection Navigator**.

Initially, this pane just shows **Connect to Team Server**. Click on this text to bring up the connection dialog. Fill in the connection details for the server you installed on and give the administrator name and password you entered during server installation. The dialog could look like this:



Team Server:	montblanc
Port:	7001
☐ Use SSL Connection	
Username:	otpc_admin
Password:	•••••••
✓ Remember Password	
✓ Connect When JDeveloper Starts	

If the connection is successful, the **Team Navigator** panel fills in with 3 headings: **Team Members**, **Work Items**, and **Versioning**. Click the little icon showing two people to bring up the context menu as shown next:



Choose **Team Administration** to open the Team Administration dialog. In this dialog, you can add your users, define teams, and connect to Work Item repositories.

Adding users and teams

It is straightforward to add new users and teams, using the **Users** and **Teams** tabs. You need to define both, because repositories are made available to teams, not users. So in order to use a repository, you have to be member of a team.

Users can be either ordinary team members or team administrators with the privilege to define, for example, team queries and team tags.

Users can be members of several teams; the drop-down box at the top of the **Team Navigator** is used to the currently active team. This might be useful if different teams use different repositories – simply change team to get access to a different set of repositories.

Connecting to a Jira repository

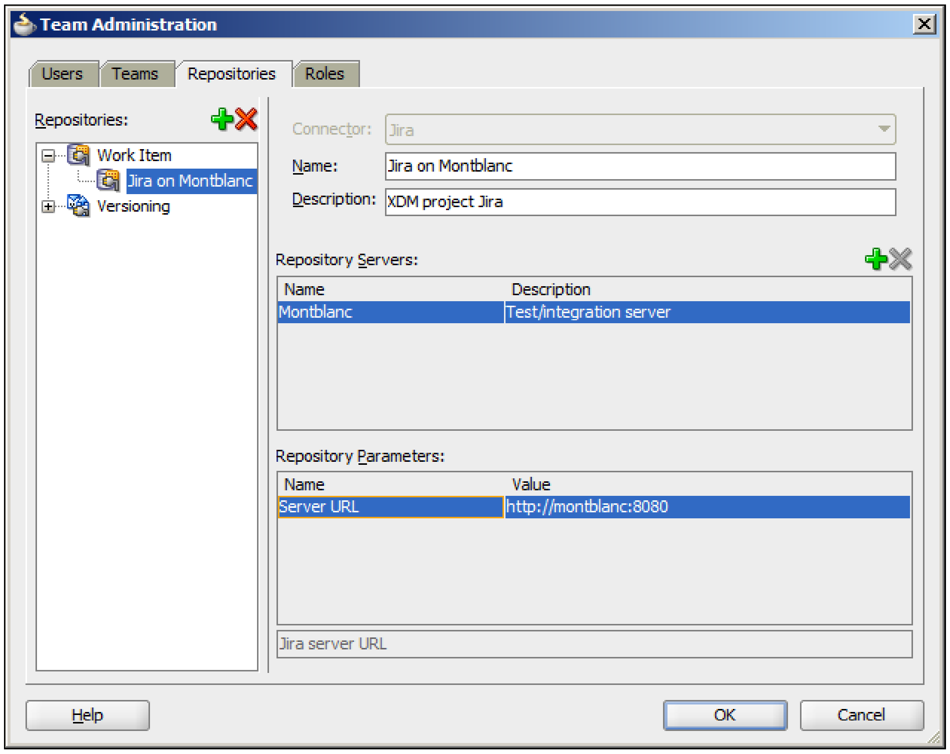
As an example of how to connect to a task repository, we will connect to a Jira repository on our development server.

No spreadsheets!

As we discussed in the last chapter, one of the tools you need for enterprise development is a bug/issue tracker. There is no excuse for spending valuable developer and project manager time on keeping lists of issues in spreadsheets.

Jira from Atlassian is a popular, user-friendly tool that integrates well with Oracle Team Productivity Center, but there are many others (Wikipedia lists over 50 issue tracking systems).

On the **Repositories** tab, select the **Work Item** node and click the green plus sign to add a new work item repository. Choose a **Connector** from the drop-down list (this list shows only the connectors you have installed). Provide a name, define a server and provide a server URL and port, as shown next:



Repositories:
- Work Item
  - Jira on Montblanc
- Versioning

Connector: Jira

Name: Jira on Montblanc

Description: XDM project Jira

Repository Servers:

Name	Description
Montblanc	Test/integration server

Repository Parameters:

Name	Value
Server URL	http://montblanc:8080

Jira server URL

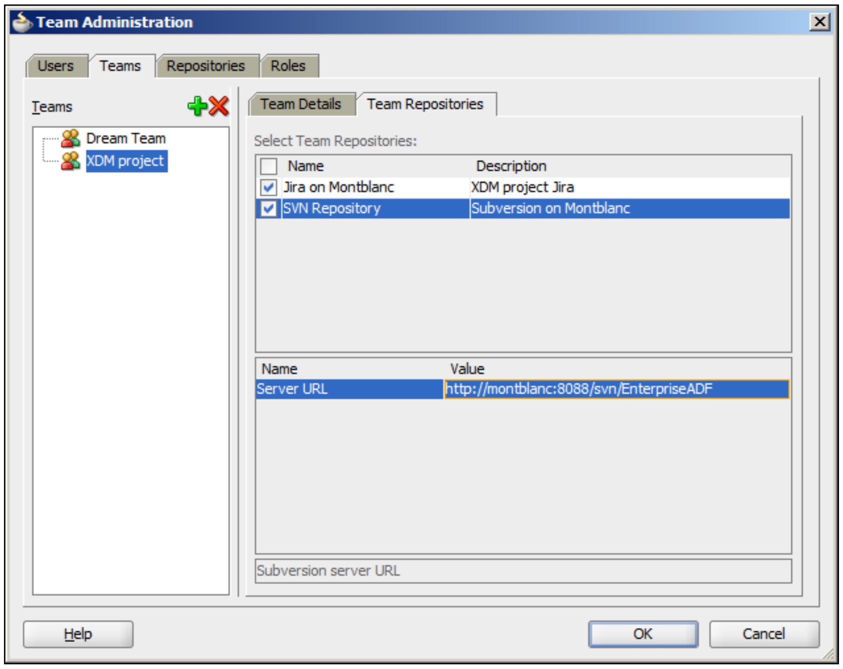
Then go into the **Teams** tab and change to the **Team Repositories** sub-tab. You need to check the checkbox for your newly created repository here in order to make it available to the team.

Connecting to a Subversion repository

To connect to your Subversion repository, select the **Repositories** tab, click the **Versioning** node and then click the green plus sign. Choose the **Subversion** connector to add a new repository, give a name and click the **Teams** tab.

The URL for a Work Item server is defined on the **Repositories** tab, but the URL for a **Versioning** server is defined on the **Team Repositories** tab.

On the **Teams** tab, change to the **Team Repositories** sub-tab and check the checkbox next to your Subversion repository. In the lower half of the tab, provide the URL for your repository (typically of the form `http://<server>:<port>/svn/<repositoryName>`):



Select Team Repositories:

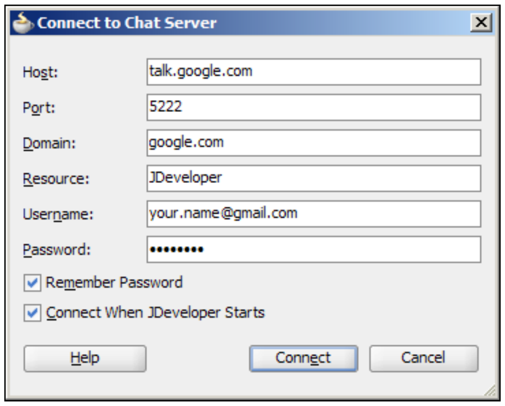
	Name	Description
✓	Jira on Montblanc	XDM project Jira
✓	SVN Repository	Subversion on Montblanc

Name	Value
Server URL	http://montblanc:8088/svn/EnterpriseADF

Subversion server URL

Connecting to a chat server

First time you choose **View** | **Team** | **Chat**, you will be prompted to connect to a chat server. To connect to Google Talk, enter the following information:



Host:	talk.google.com
Port:	5222
Domain:	google.com
Resource:	JDeveloper
Username:	your.name@gmail.com
Password:	••••••••

✓ Remember Password

✓ Connect When JDeveloper Starts

Disconnecting

Once you are done with the management tasks, click on the team icon at the top of the **Team Navigator** pane and choose **Disconnect** from the pop-up menu. The **Team Navigator** changes back to the **Connect to Team Server** heading. Click on this heading and connect back to the Team Server, this time as yourself (not using the administrator account).

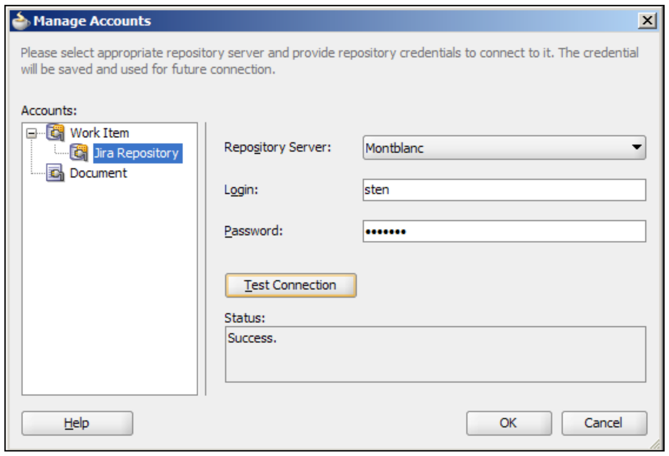
Getting started with work items

When starting a project, the project manager should convert the project plan into a list of tasks for each developer, and register these as work items in your bug/issue/task tracking system. This allows each developer to write comments and register progress on tasks, split tasks into more manageable subtasks and even re-assign tasks to other team members.

Connecting to your work item repository

Once you are logged in to your own account in Team Productivity Center, expand the **Work Items** heading by clicking on the little triangle. You see the work item repositories defined for your currently selected team. If you are a member of several teams, select your current team from the drop-down list at the top of the **Team Navigator**.

The first time you click on the plus icon to expand a repository, you will probably get a message saying "Failed to log in to ...." This just means that you have not yet defined your credentials for that repository. Simply click **Yes** to go to the **Account Manager**, enter your username and password for that repository, test the connection and click **OK**:



Please select appropriate repository server and provide repository credentials to connect to it. The credential will be saved and used for future connection.

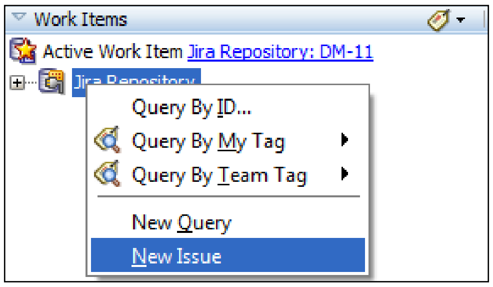
Repository Server: Montblanc
Login: sten
Password: •••••••
Status: Success.

Oracle Team Productivity Center stores your credentials for work item repositories securely. If you have several repositories, you only need to log on to the Team Productivity Center to get access to all of them.

Because JDeveloper is already storing your version control (for example, Subversion) credentials, Oracle Team Productivity Center does *not* store these. This means that when you log out of Team Productivity Center, you log out from your work item repositories, but not from your versioning repository.

Creating a work item

To create a new work item, right-click on the repository and choose **New Issue** (or **New Task**, or **New Bug**, depending on your repository). You get a new tab in the main JDeveloper window where you can enter details about your Issue, Task or Bug:



The fields on the tab depend on the repository and the connector – in some cases, the underlying repository might offer more functionality through its native interface than through the connector. This can be caused both by limitations in the API the repository offers and limitations in how much functionality the provider of the connector has decided to offer.

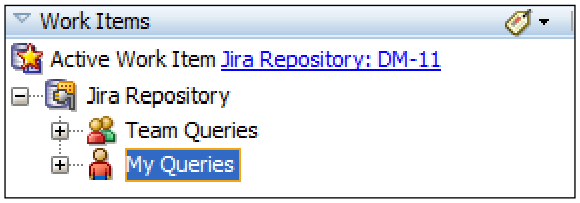
Daily work with work items

As a developer, your daily work with work items will include finding, updating, linking and tagging them.

Finding work items

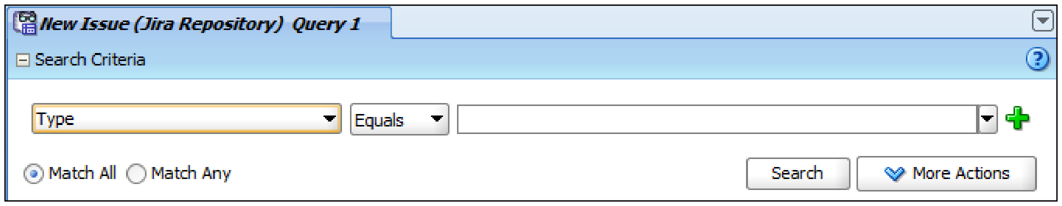
Your project manager has probably already created some tasks for you, based on the project plan. Depending on the level of detail you want, you might have a complete use case as a task (for example, "UC 008 Task Overview and Edit"), or you might have a detailed breakdown with multiple tasks (for example, "UC 008 Task flow", "UC 008 View objects", and so on).

To find your assigned work items, you open the work item repository and perform a query. Under the repository, you will see two nodes:



Under **Team Queries**, you find the queries your team administrator has defined for you, and under **My Queries**, you can define and run personal queries.

To perform a general query, you can right-click on the **My Queries** node and choose **New Query**. A **New Issue Query** tab opens (or **New Task Query**, or **New Bug Query**, depending on your repository):



Here, you select an attribute from the first drop-down box, a condition from the second and select or type a value in the third field. For more complicated queries, you can click the plus sign to add additional criteria and choose either **Match All** or **Match Any**. When you click **Search**, your query is executed. If you want to save the query for later use, click the **More Action** button and then choose **Save As**. If you are a team administrator, you have the possibility to save the query for the whole team or just for yourself.

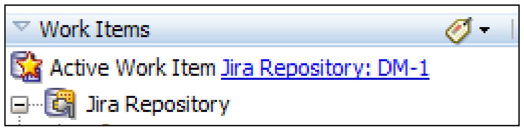
If you know the ID of the issue you are looking for, you can simply right-click on the repository, choose **Query by ID** and enter a work item ID. The work item opens in the JDeveloper main window as a new tab. You can change it as necessary and then close it like any other tab by using the little **x** icon in the tab title.

You can query work items by tag as well. Right-click on the repository or the tag icon next to the **Work Items** header, choose **Query By ... Tag** and choose a tag. You get a list of all items with that tag. From this list, you can double-click a work item to open it, or click the **Delete** icon to remove the tag from the item.

To execute a saved query, find it under either the **My Queries** or **Team Queries** node under the repository, right-click and choose **Run**.

Setting the active work item

When you have queried an item, you can right-click on it and choose **Make Active** to make this work item your **active item**. When you have an item on screen, you can click on the **Make Active** button at the top to set that item as your current or active item. The active work item is shown at the top of the **Work Items** list:



Whenever you commit your changes to Subversion, Oracle Team Productivity Center will by default associate your Subversion commit with the active work item. This is described in more detail next in the section on working with both Subversion and Oracle Team Productivity Center.

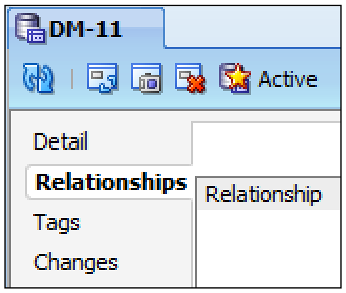
Stay focused

Always keep an active work item selected. This serves as a subtle hint to your brain to stay at the task at hand and increases your productivity.

Linking work items

It is possible to link work items to each other—for example, a bug in Bugzilla could be linked to a task in Microsoft Project Server.

To create and view these links, you use the **Relationships** sub-tab to the left of the item itself:

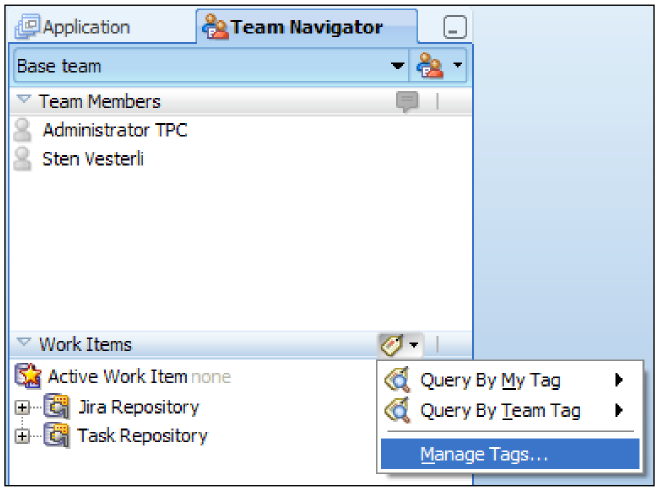


Here, you can see the relationships the work item is part of, and add new ones. If you activate this sub-tab, you have the option to link the active work item to another open work item or an item you have tagged.

Tagging work items

You can select a number of work items by **tagging** them. You are free to define your own tags with whatever meaning you want—this could be today's work, stuff for the next release, items to discuss with Michael, or anything else.

To define tags, you click on the little tag icon at the top of the **Work Items** section of the **Team Navigator**:



Click **Manage Tags** to define the tags you want. If you are a team administrator, you can also define tags available to the whole team.

You can add a tag to an item when you first create it or anytime later you have the item open. Simply click on the **Tags** sub-tab on the left side of the work item tab to add and remove tags.

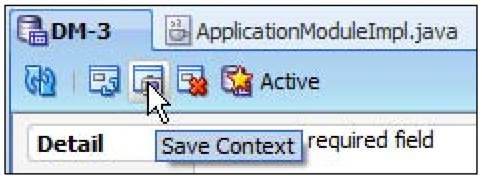
Chatting with team members

You can double-click on a team member in the **Team Members** part of the **Team Navigator** to start a chat with that person (in the **Chat** pane).

Saving and restoring context

Remember the discussion from the beginning of the chapter about the cost of context switching? Oracle Team Productivity Center contains a very nice feature to minimize the cost of context switching: The ability to save and restore context.

To use this feature, open a work item and click the **Save Context** button in the button list at the top of the work item:



This will save your current context, including information about all open files in the main window, the currently active workspace and the state of the **Application Navigator**. When your boss walks up to you with an urgent task that requires you to drop everything, you can simply:

1. Save your JDeveloper context.
2. Do the urgent task and return to JDeveloper.
3. Click **Restore Context** (the button to the left of **Save Context**).

JDeveloper will open all the files that were open when you saved context, and will open the application workspace you were working on, exactly as you left it.

The Boss Key

Many years ago, before multi-tasking operating systems were common, computer games would include a "Boss key" – a special keystroke that would hide the game and show something that looked like a work-related document or spreadsheet. The **Save Context** button is JDeveloper's "Boss Key" – not used to hide the application, but to make sure you can get back to where you were as soon as you are done with the boss' urgent task.

Version control

As we discussed in *Chapter 3*, *Getting Organized*, you absolutely, definitely need to use some kind of version control system in an enterprise development effort. You should even use it if you are the only developer on your team!

Oracle JDeveloper comes with extensions for many different source control systems that you can download and install using the **Help | Check for Updates** functionality. Supported systems include CVS, Perforce, ClearCase and many more.

The Subversion software

In this book, Subversion will be used as an example. Subversion is very popular among JDeveloper users, for several reasons:

- It is widely used – lots of other people are using it and many other tools can read your Subversion repository
- It is free – always a good point
- It is well integrated into JDeveloper
- It is atomic – either your whole commit goes into the repository or nothing does. Since ADF projects consist of many interdependent files, this is very much desirable

To use Subversion, you need a Subversion server and a client. The Subversion server is available for all platforms—if your version control server is based on Microsoft Windows, you can use the VisualSVN (`http://www.visualsvn.com/server`), which comes as a standard Windows .MSI install file.

JDeveloper comes with a Subversion client for working with code, but if you keep other files in Subversion as well, you probably want a stand-alone client to update and commit to the repository. A very popular (and free) client is TortoiseSVN (`http://tortoisesvn.tigris.org`), which integrates directly into the Windows Explorer context (right-click) menu.

Effective Subversion

If you have never used Subversion before, you need to familiarize yourself with the tool. There are many excellent resources available on the internet—a good place to start is *Chapter 1*, *The ADF Proof of Concept* of the free e-Book on Subversion (`http://svnbook.red-bean.com`).

There are many ways to use a tool like Subversion, but for an enterprise ADF project, you should:

- Use the standard structure of **trunk**, **tags** and **branches**:
    - `trunk` is where you keep your main development code.
    - `branches` is where you store variations of the code. Each branch is a copy of the code as it looked when you created the branch.
    - `tags` is where you store copies of your code corresponding to a specific tag. You typically create a tag for each released version of your code.

Efficient storage

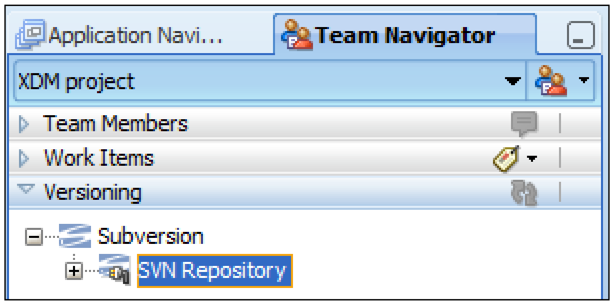
Even if it looks like Subversion keeps many copies of your code, the branches and tags copies do not take up any significant space—they are only pointers or *virtual copies*.

- In your trunk, create folders for each application workspace with the same name as your application workspace
- Use Copy-Modify-Merge (in general, do not lock files)
- Check out and commit at the application workspace level (**Commit Working Copy**, not just **Commit** for a single file)

Logging on

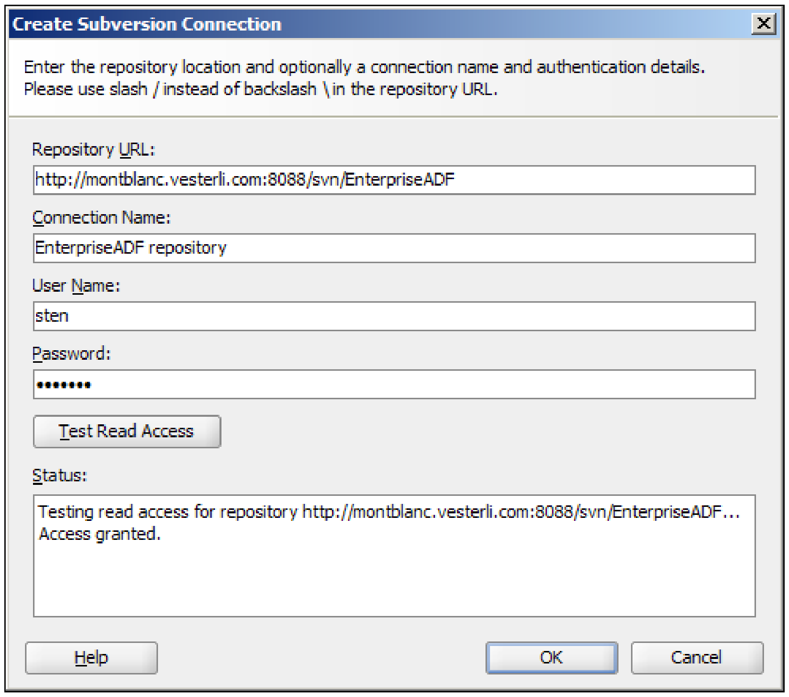
Before you start working with Subversion (or another versioning repository), you need to log on:

If you are using Oracle Team Productivity Center, you can see your team repository under the **Versioning** header in the **Team Navigator**. Double-click on the connection to open the **Edit Subversion Connection** dialog, where you can enter your username and password:



If you are not using Oracle Team Productivity Center, you can choose **View | Team | Versioning Navigator** to display the versioning navigator. Here, you can right-click on the node for your versioning system (for example, **Subversion**) and choose **New Repository Connection** to bring up the **Create Subversion Connection** dialog for defining a repository URL, username, and password.

The first time you import an application workspace into Subversion, you will be prompted to create a connection as shown next:



Enter the repository location and optionally a connection name and authentication details.
Please use slash / instead of backslash \ in the repository URL.

Repository URL: http://montblanc.vesterli.com:8088/svn/EnterpriseADF

Connection Name: EnterpriseADF repository

User Name: sten

Password: •••••••

Status:

Testing read access for repository http://montblanc.vesterli.com:8088/svn/EnterpriseADF...
Access granted.

If you are working with Oracle Team Productivity Center, you already have the URL—otherwise you will have to write it here. Provide a connection name and credentials to log in to Subversion and click **Test Read Access**. Once you see an "Access granted" message, you know that the connection information is valid and you can click **OK**.

Initial load

As soon as you have created the application workspace and the first few functioning objects, add your application to the Subversion repository (in Subversion terminology, this is called *importing*).

To import your application workspace, select a project in the workspace and choose **Versioning | Version Application**. If you have more than one connection to a versioning system, JDeveloper prompts you to select one of them.

The Import to Subversion wizard starts and guides you through the initial import:

1. In the Destination step, choose **trunk** and then click on the little "Add folder" icon above the path box to the right. In the **Create Remote Directory** dialog, enter the name of your application workspace as directory name and provide a comment.
2. In the **Source** step, check that the **Source** directory is your application workspace directory. Provide a comment like **Initial import** and click **Next**.
3. You do not need to change anything in the **Filters** step.
4. In the **Options** step, leave **Perform Checkout** checked to immediately check out your application to continue working
5. In the **Summary** step, review the options and click **Finish**.

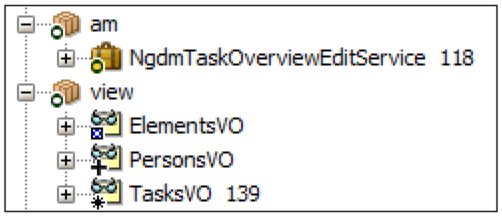
You can see your import running in the **SVN Console Log** window.

Once the import is complete, you will notice the following changes in the **Application Navigator**:

- The projects now show the repository server they came from.
- Each file now has an indicator showing its state. Right now, all of them have status **Unmodified**, but you will see this change as you and your team members work on the project.
- Each file now shows a version number. Do not worry about the numbering – they will seem to increase in uneven jumps. In fact, the subversion version number increases every time anybody on the project commits anything. Refer to a Subversion book (for example, the free e-Book at `http://svnbook.red-bean.com`) for a more detailed explanation.

Working with Subversion

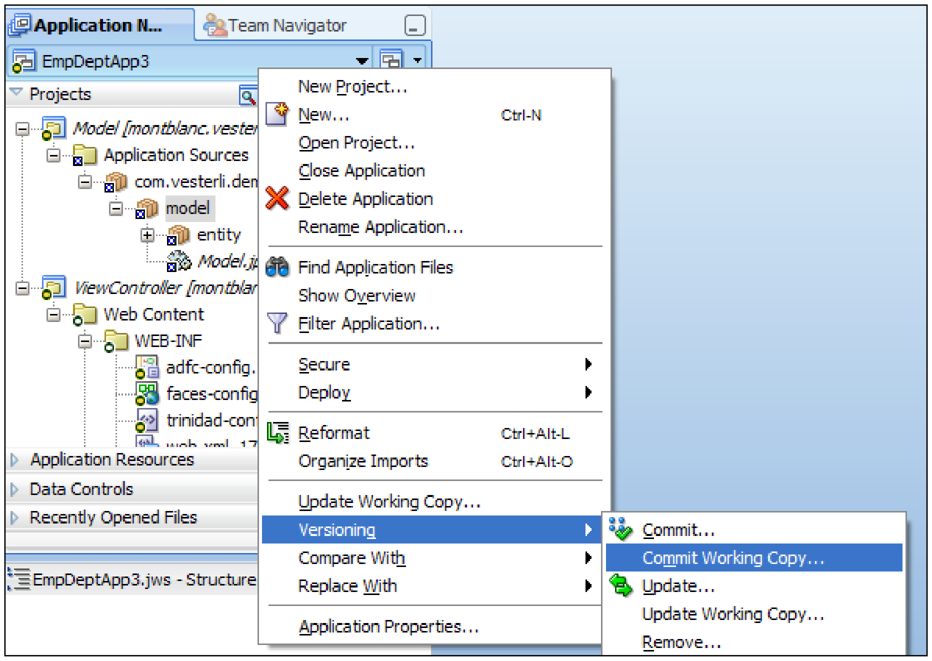
As you work with your project, you will notice different icons on your files and directories.



The little circle with the yellow center indicates a versioned object that has not been changed since the last commit, and the object marked with an asterisk (**TasksVO** above) is an object that has been changed since last commit. The file with a little white X in a box (**ElementsVO** above) is a new, unversioned file, and the file with the plus sign (**PersonsVO** above) is a new file that is scheduled for addition on next commit.

You can also choose **Versioning | Pending Changes** to call up the **Pending Changes** window showing files changed locally ("outgoing"), new files ("candidates" to be placed under version control, and files changed by other users on the server, but not yet applied to your working copy ("incoming").

Every time you have made a significant change (and tested it, of course), you check your changes into the central repository by right-clicking on the application workspace title in the **Application Navigator** and choosing **Versioning | Commit Working Copy**:



If you are only using Subversion (and not Oracle Team Productivity Center), the **Commit Working Copy** only prompts you for a commit comment. If you are using Oracle Team Productivity Center together with Subversion, the dialog is different—see the following section.

Comment Templates

If you like, you can configure a number of standard comments as "Comment Templates" under **Tools | Preferences | Versioning | Comment Templates.**

When you click OK, you see the Subversion commands issued to commit your changes in the **SVN Console Log** window. The icons in the **Application Navigator** also change, and the **Pending Changes** window should now be empty.

In order to make JDeveloper automatically add new files when you commit your working copy, you need to set another JDeveloper preference. Under **Tools | Preferences | Versioning | General**, check the checkbox **Automatically Add New Files on Committing Working Copy**. If you want to manually control when new files are added to Subversion, you can leave this checkbox unchecked and then explicitly add them from the **Pending Changes** window on the **Candidates** tab. Clicking on the green plus sign on this tab will change the file status from **Not Versioned** to **Scheduled for Addition**. This means that they will be imported into Subversion next time you commit your working copy.

Teamwork with Subversion

If you have structured your application as described in *Chapter 3*, *Getting Organized*, you will have a number of different application workspaces. For some workspaces, you might be the only developer, but for most, there will be a small team of developers working on the code at the same time.

Getting a new copy

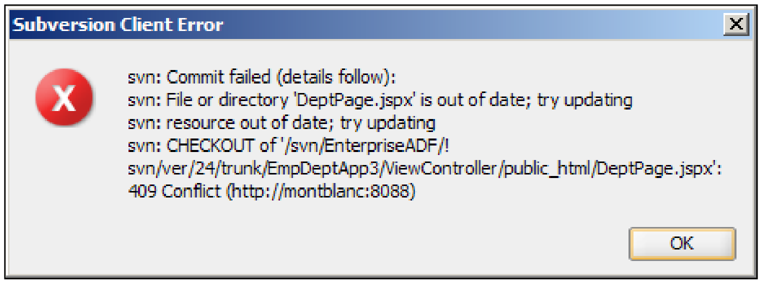
When another team member needs to start working with an application workspace already imported into Subversion, that person simply chooses **Versioning | Check Out**. He then defines a Subversion connection using his Subversion credentials and navigates to the application workspace folder under the trunk of the version tree. JDeveloper will now get the latest version to the local machine as that person's working copy.

Getting other people's changes

When your colleague commits new files to the application workspace, these files will show up in the **Pending Changes** window as "incoming." To get these files, you right-click on the application in the Application Navigator and choose **Update Working Copy**. This will bring your working copy up to date with the changes made by other developers.

Automatic merge

Occasionally, you and another team member will both have made changes to the same file. Of course, Subversion does not just allow the second developer to overwrite the changes made by the first. Instead, if your colleague committed her change first, you will get a message from Subversion when you try to submit your file, like this:



```
svn: Commit failed (details follow):
svn: File or directory 'DeptPage.jspx' is out of date; try updating
svn: resource out of date; try updating
svn: CHECKOUT of '/svn/EnterpriseADF/!
svn/ver/24/trunk/EmpDeptApp3/ViewController/public_html/DeptPage.jspx':
409 Conflict (http://montblanc:8088)
```

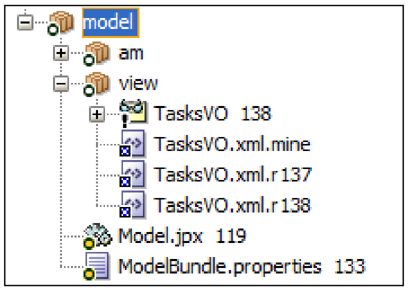
This message tells you that the file has changed in the repository since you originally retrieved it. If you look at the **Pending Changes** window, you will also see that the same file is listed both as "Outgoing" (with your change in it) and as "Incoming" (with the other developer's change in it).

If you now perform an **Update Working Copy**, Subversion is clever enough to automatically merge your change and the incoming change to a new file containing both your updates. You can see the changes by right-clicking on the conflicting file and selecting **Compare With** – this allows you to compare all versions of the file.

Handling conflicts

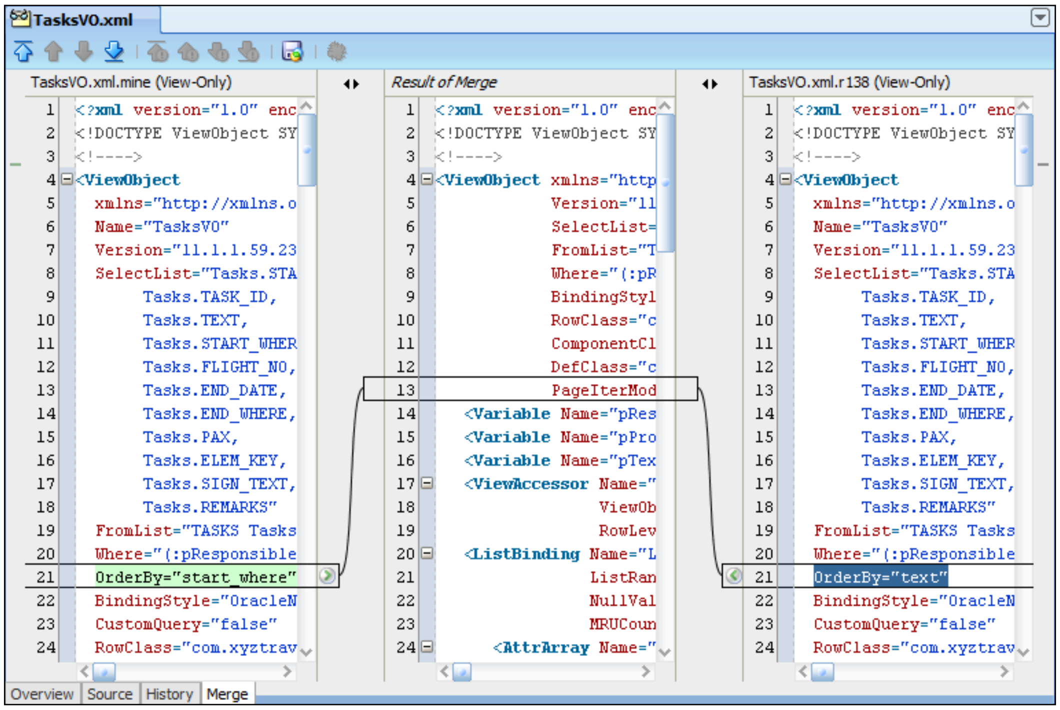
But what happens if your colleague has changed some code, and you then change *the exact same lines of code* in your working copy? Initially, you will get the "try updating" message. When you try updating, you will discover several things happening:

- You suddenly have four versions of the file in question in the Application Navigator—one of them with an exclamation mark, indicating a conflict:



- The SVN Console Log will show a conflict
- The **Pending Changes** window will show your outgoing file with status "Conflicts"

Since Subversion does not pass judgment on which change is the better, you and your colleague will have to decide how to handle the conflict. Fortunately, JDeveloper has a very nice graphical interface to help you implement your decided conflict resolution. Simply right-click on the conflicted file and choose **Resolve Conflicts**. This brings up the file in merge mode – showing your version on the left (the `.mine` file), the other person's code on the right, and an editable, final version in the middle. You can use the **>** and **<** buttons to move code from either side into the middle, or you can simply edit the central file, if the result of your merge contains parts from both sides:



```
<?xml version="1.0" enc
<!DOCTYPE ViewObject SY
<!---->
<ViewObject
  xmlns="http://xmlns.o
  Name="TasksVO"
  Version="11.1.1.59.23
  SelectList="Tasks.STA
       Tasks.TASK_ID,
       Tasks.TEXT,
       Tasks.START_WHER
       Tasks.FLIGHT_NO,
       Tasks.END_DATE,
       Tasks.END_WHERE,
       Tasks.PAX,
       Tasks.ELEM_KEY,
       Tasks.SIGN_TEXT,
       Tasks.REMARKS"
  FromList="TASKS Tasks
  Where="(:pResponsible
  OrderBy="start_where"
  BindingStyle="OracleN
  CustomQuery="false"
  RowClass="com.xyztrav
```

```
<?xml version="1.0" enc
<!DOCTYPE ViewObject SY
<!---->
<ViewObject xmlns="http
            Version="11
            SelectList=
            FromList="T
            Where="(:pR
            BindingStyl
            RowClass="c
            ComponentCl
            DefClass="c
            PageIterMod
   <Variable Name="pRes
   <Variable Name="pPro
   <Variable Name="pTex
   <ViewAccessor Name="
                 ViewOb
                 RowLev
   <ListBinding Name="L
                ListRan
                NullVal
                MRUCoun
     <AttrArray Name="
```

```
<?xml version="1.0" enc
<!DOCTYPE ViewObject SY
<!---->
<ViewObject
  xmlns="http://xmlns.o
  Name="TasksVO"
  Version="11.1.1.59.23
  SelectList="Tasks.STA
       Tasks.TASK_ID,
       Tasks.TEXT,
       Tasks.START_WHER
       Tasks.FLIGHT_NO,
       Tasks.END_DATE,
       Tasks.END_WHERE,
       Tasks.PAX,
       Tasks.ELEM_KEY,
       Tasks.SIGN_TEXT,
       Tasks.REMARKS"
  FromList="TASKS Tasks
  Where="(:pResponsible
  OrderBy="text"
  BindingStyle="OracleN
  CustomQuery="false"
  RowClass="com.xyztrav
```

When you are done merging, click the **Save and Complete Merge** button in the toolbar above the files (a little diskette icon with two colored circles). This completes the merge, and you can now commit your working copy to the repository.

After committing, you can click on the little refresh button (two blue arrows in a circle). You will see that the extra versions of the file (the two `.rXXX` files and the `.mine` file) are now gone, indicating successful conflict resolution.

Avoiding conflicts

It takes time to resolve conflicts, so you should make an effort to avoid them. There are both "soft" and "hard" methods you can use for this. The "soft" methods are procedural (ways you agree to work in your team) and the "hard" methods are technical, implemented using the capabilities of your version control tool.

The first "soft" method is to take the approach recommended in the last chapter and split your application into a number of separate workspaces, connected via ADF Libraries, you have already reduced the risk of version conflicts significantly.

The second "soft" method you can use is to assign owners to the files that are often the source of version conflicts. If you decide that only Karen is allowed to commit changes to the two files that the whole team keeps fighting over, she can keep the purpose and entire structure clear in her mind. If anybody else needs a change made, they can tell Karen and let her make the change.

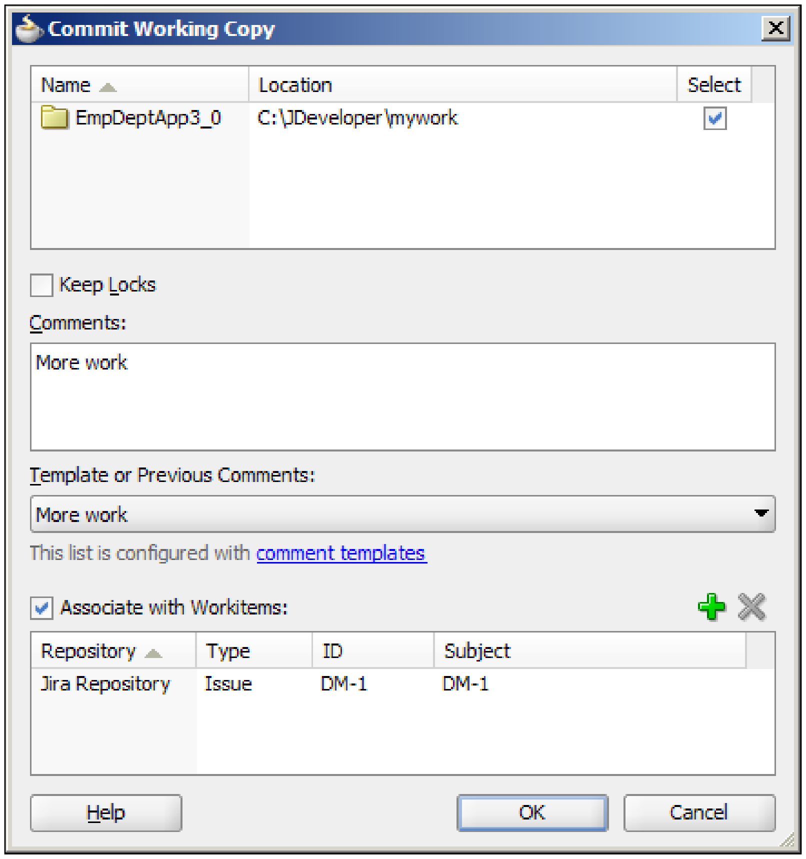
Unfortunately, some of the files that you might want to assign an owner to are files are updated by many different ADF wizards. In this case, a developer might not even realize that he is making a change to a contested file. To avoid this, your can use a "hard" method: Explicitly locking files. If you are using Subversion, you can right-click on a file and choose **Versioning | Lock** from the context menu. This will prevent anybody else from checking in a change to the locked file. Use this method with caution – you can seriously disrupt the work of other developers if you lock important ADF files.

Unmergeable files

Some files cannot be merged; you will have to lock these files to avoid wasted work. Examples of unmergeable files are JDeveloper diagrams, word processing files, spreadsheets, and images.

Subversion and Oracle Team Productivity Center together

When you install Oracle Team Productivity Center, the **Commit Working Copy** dialog changes to look as shown next:



Name	Location	Select
EmpDeptApp3_0	C:\JDeveloper\mywork	✓

Comments:

More work

Template or Previous Comments:

More work

This list is configured with comment templates

Repository	Type	ID	Subject
Jira Repository	Issue	DM-1	DM-1

Notice the **Associate with Workitems** box at the bottom. Here, you can choose the work items that this Subversion commit is related to. By default, your active work item will be associated. You can uncheck the checkbox to associate nothing, or you can click the green plus sign to add additional work items related to this commit.

Once you click **OK** to perform the commit, Oracle Team Productivity Center registers the association. The next time you open the work item, you will see the Subversion commit listed on the **Changes** sub-tab. This allows you to track exactly what was done to address a specific issue – useful for example if you have to revisit the issue later.

Subversion, Jira and FishEye together

If you are using Jira and FishEye (a code inspection tool, also from Atlassian), you should always make sure to enter the Issue ID in the Comments field when committing your changes to Subversion. Right now, this is not automatic – but it might become an option in a future release of Oracle Team Productivity Center.

The Issue ID is the key that FishEye uses to link a Subversion commit to a Jira Issue.

If you wish to write your own reports against Oracle Team Productivity Center, you can look at the database schema where you installed the tool. One of the tables in this schema is called **OTPC_WORKITEM_CHANGE**, and it contains the association of work items with code commits.

Summary

You have installed Oracle Team Productivity Center and integrated it with the new Jira issue repository you will be using for the XDM project as well as the Subversion repository where you will be storing all your code.

Everybody on the team has been defined as a user in Oracle Team Productivity Center so it was easy to connect each development workstation to Jira and Subversion. Each developer can now work directly with your Jira work items from within JDeveloper, and can easily get new code from Subversion and commit changes.

Back from the excellent kick-off pizza party last night, you and the rest of the XDM team are ready to start writing the first production code for DMC Solutions' brand-new enterprise application. Let's move on to the next chapter and get started!

5
Prepare to Build

If you have children, you might be making gingerbread figures for Christmas. Your artistic children are likely to be hand-crafting each figure individually—but you, as an efficiency-oriented adult, are probably using a cookie cutter to produce a whole batch of almost identical Santas.

When you need to create many similar objects efficiently, you use a template (like your cookie cutter). And when you need to create dozens of task flows and hundreds of screens, all with some common elements, you use the ADF cookie cutters: Task Flow Templates and Page Templates for the frontend View part, and Framework Extension Classes for the backend Business Component part.

Task flow templates

You will be using bounded task flows to implement the use cases or user stories in your application, so you will probably be building quite a few of them. And they are likely to share some common functionality like error handling.

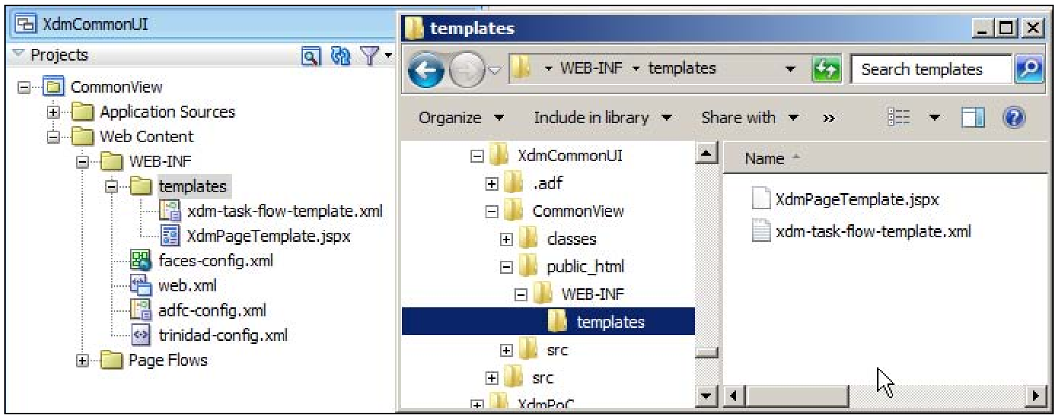
Therefore, you should base all your bounded task flows on **Task Flow Templates** that can contain all the common functionality.

Always use task flow templates

Even if you do not know of any common functionality you might want to use in all your bounded task flows, base them on a template anyway. In that way, you have the possibility to add something later if you find a need during development.

Task flow templates can be nested within one another, so you can even create a master task flow template and then later produce other, more detailed templates based on this master.

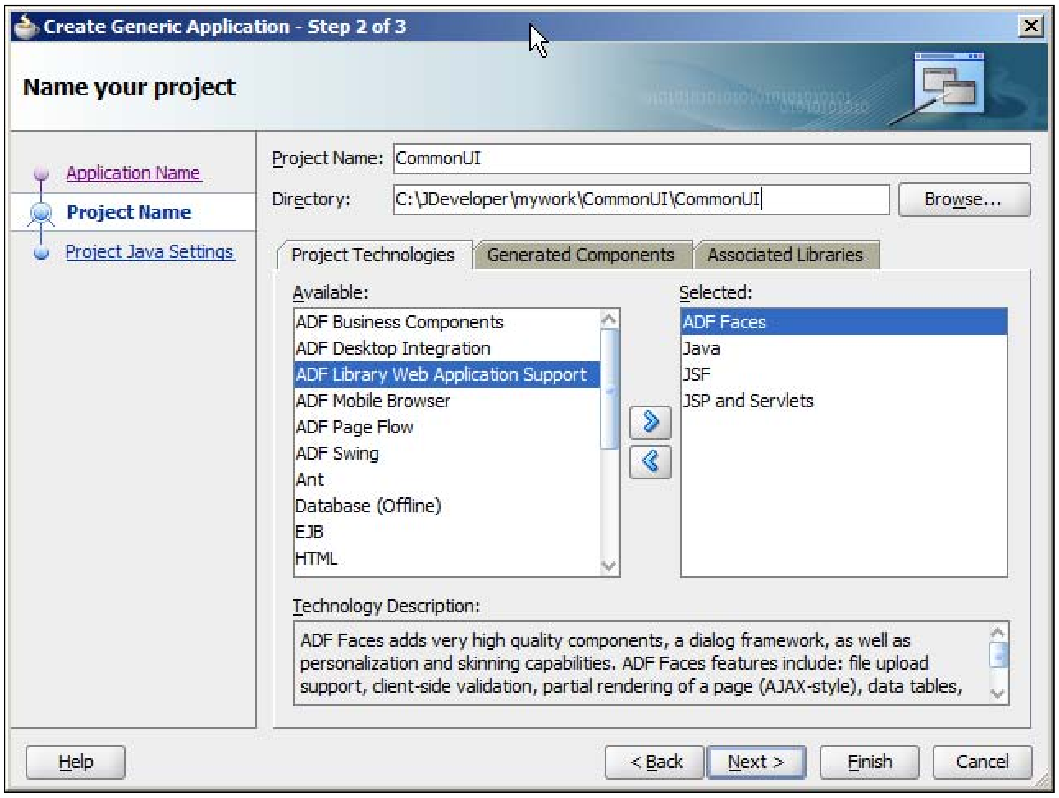
Because templates do not need to be directly accessible to the end user of the application, you should take advantage of the fact that application server by default hides the content of the `WEB-INF` directory. Create a subdirectory called `templates` under this directory and place your templates there:



The node in the **Application Navigator** shown as **Web Content** corresponds to the directory `public_html` in the file system.

Creating a task flow template

If you use the structure recommended in *Chapter 3*, *Getting Organized*, your task flow template should go into your CommonUI workspace. If you have not already created this, choose **File | New** and then **Generic Application**. Give your workspace the name **CommonUI** and set the **Application Package Prefix** to your project base package (for example, `com.dmcsol.xdm`). In Step 2 of the wizard, give your project the name **CommonUI** and choose to include **ADF Faces** technology:



Project Name: CommonUI

Directory: C:\JDeveloper\mywork\CommonUI\CommonUI

Available:

ADF Business Components
ADF Desktop Integration
ADF Library Web Application Support
ADF Mobile Browser
ADF Page Flow
ADF Swing
Ant
Database (Offline)
EJB
HTML

Selected:

ADF Faces
Java
JSF
JSP and Servlets

Technology Description:

ADF Faces adds very high quality components, a dialog framework, as well as personalization and skinning capabilities. ADF Faces features include: file upload support, client-side validation, partial rendering of a page (AJAX-style), data tables,

In Step 3 of the wizard, set the **Default Package** to the sub-package `view` under your project base package (for example, `com.dmcsol.xdm.view`).

To create a task flow template, select your **CommonUI** project and choose **File | New** and then **Web Tier | JSF | ADF Task Flow Template**. Give your template a name and a location (under `WEB-INF` as described above). You will normally check the checkbox **Use Page Fragments**. Remember that we use page fragments in dynamic regions on pages—this allows us to swap one page fragment out and another one in, giving your ADF application a "desktop application" feel. Do not choose **Create Train** for your master task flow template, because not everything is going to be a train.

Contents of your master task flow template

For your top-level or master task flow template, consider the following elements:

- A common exception handling page
- Common help or about pages
- Initializer and finalizer code

Exception handling page

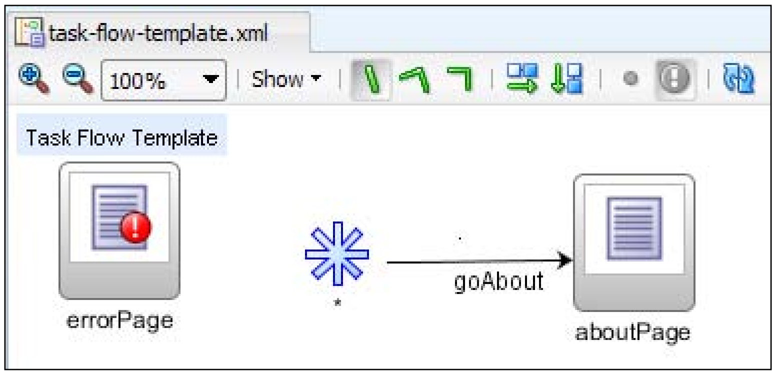
Each task flow can have one exception handling page – the page automatically invoked by the ADF framework in case of unhandled exceptions in your code. The exception handling page is marked with a red exclamation mark, and you can define a page as the exception page by selecting it in the diagram and clicking the red exclamation mark icon in the toolbar above the diagram.

Always have an exception page

You should have a well-designed and friendly exception page – never show your users the ugly Java exception stack.

Common Help or About pages

If you plan for your application to have a common **About** or **Help** page, this should also go into the master task flow template. Additionally, you should place a **Wildcard Control Flow Rule** on the diagram with a transition to your about or help page. The wildcard control flow rule looks like a blue asterisk and can be found under **Control Flow** in the **Components** section of the Component Palette.



The purpose of the wildcard control flow rule is to make a transition available to all pages in the task flow without cluttering the diagram with a lot of lines. You can name it something other than just * to limit the pages that can invoke the transition – refer to the online help for details.

Initializers and finalizers

If you want to run specific code before and after a task flow executes, write Java methods doing what you want, and then refer to these in the task flow (view the task flow in **Overview** mode instead of **Diagram** mode and choose the **General** sub-tab).

You might have to initialize managed beans, you might want to store session information, or you might want common logging code to be executed whenever entering and exiting task flows. If you base your task flows on a task flow template, you have a place to add this code if you want.

Creating several levels of templates

If you know that your application will contain a number of specific sub-types of task flows, you can create a task flow template hierarchy consisting of a master task flow template and a number of more detailed task flow templates. When creating detailed task flows, you choose to base them on your master task flow template in the **Create Task Flow Template** dialog.

Your structure becomes easiest to manage if you only define each element (for example, the exception handling page) at one level. However, it is possible to define, for example, the exception handing page in both the master and the detail templates. Reasonably enough, the detail takes precedence over the master task flow template.

Using Task Flow templates

When you create a task flow based on a template, pay attention to the checkbox **Update the Task Flow when the Template Changes**. This one should **always be checked** because this *links* your task flow to the template. If this checkbox is not checked, your new task flow will receive a *copy* of the elements in the template, but later changes to the template will not affect existing task flows. Normally, you want changes to your template to affect all task flows based on the template, so be sure to check this checkbox.

Page templates

Just like you should never build at task flow without basing it on a task flow template, you should never build a page without basing it on a **page template**. Page templates are always referenced (never copied), so any change you make to a page template will affect all pages based on the template.

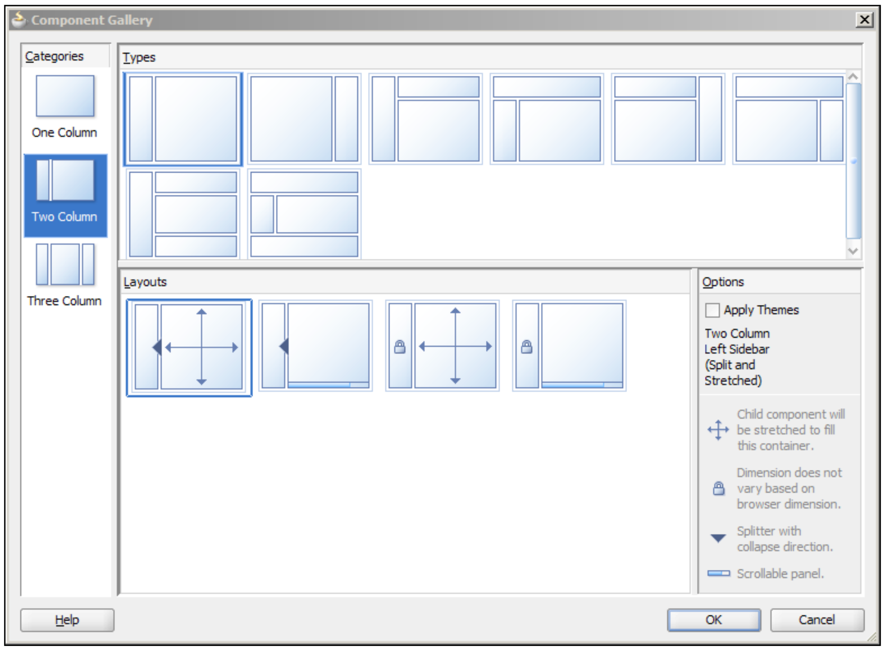
JDeveloper comes with two advanced templates called **Oracle Three Column Layout** and **Oracle Dynamic Tabs Shell** that you can look at to see examples of what an enterprise template might look like.

Creating a page template

To create a page template, open the CommonUI project in your common user interface workspace. Choose **File | New** and then **Web Tier | JSF | JSF Page Template**. The **Create JSF Page Template** page opens.

If you are already a JSF layout wizard with several projects under your belt, you will probably want to create your layout from scratch, using layout components from the component palette.

However, you are not already familiar with JSF page layout, you can take a shortcut by checking the **Use a Quick Start Layout** checkbox and click **Browse** to see the built-in quick start layouts as shown next:


Component Gallery
Categories
One Column
Two Column
Three Column
Types
Layouts
Options
Apply Themes
Two Column Left Sidebar (Split and Stretched)
Child component will be stretched to fill this container.
Dimension does not vary based on browser dimension.
Splitter with collapse direction.
Scrollable panel.
Help
OK
Cancel

As part of your requirements gathering process, you should have produced sketches of your user interface, so you already know which layout you need. Start by selecting a one, two or three column layout in the left-hand column, then choose a type and finally choose a Layout. You should use a layout where your main area is stretchable (one with the crossed arrows symbol) in order to make the best use of the available browser area.

The layout you choose here is not fixed – it is simply a good starting point that you can further customize later as you gain experience with JSF layout.

Using layout containers

In order to make the most of the available screen area, you should always start your layout with a stretchable container like a **Panel Stretch Layout** or a **Panel Splitter**.

Stretching exercises

Some layout components (for example, **Panel Stretch Layout** and **Panel Splitter**) stretch themselves and stretch the components inside them. Other layout components (for example, **Panel Group Layout** and **Panel Form Layout**) stretch, but do not stretch the components inside them.

Inside your outer container, use other layout containers as necessary until you achieve the layout you wish. It is often necessary to place layout containers inside other layout containers several times in order achieve the layout you want.

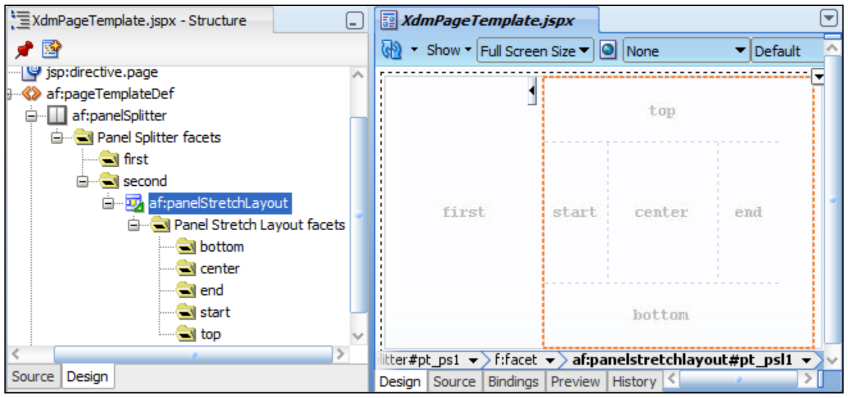
Since normal input fields cannot stretch, they should only placed inside containers that do not stretch the components they contain. Components that can stretch (for example, a **Table** component) should be placed inside a container that does stretch the components they contain.

Facet definitions

In JSF terminology, a **facet** is an area where components can be placed. Facets are used in two places:

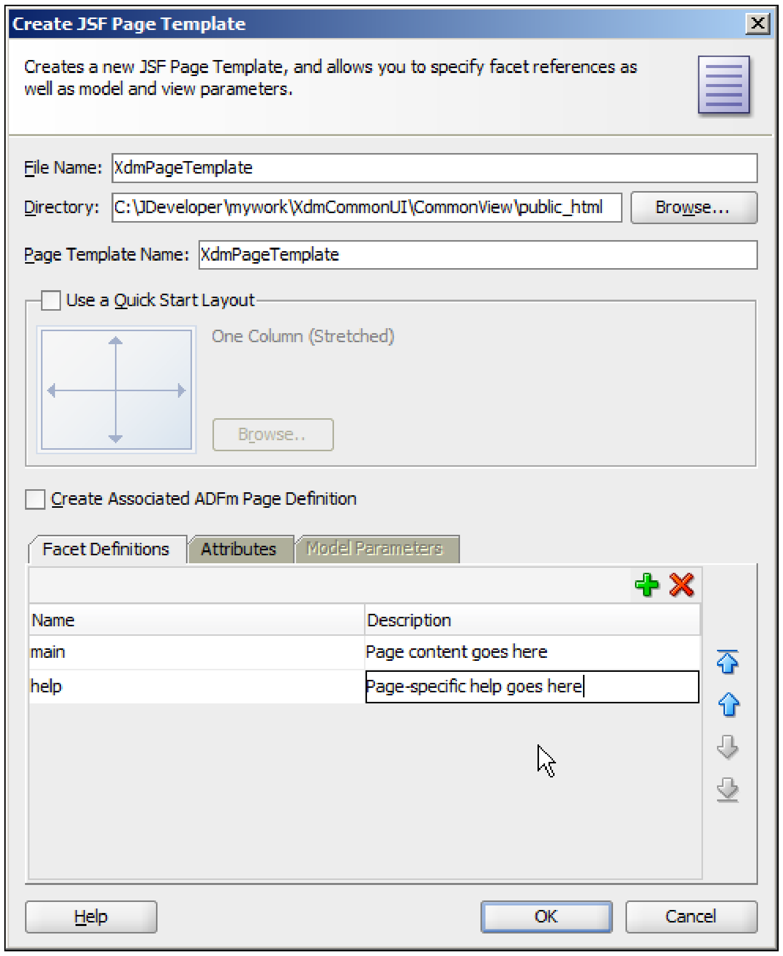
- The JSF layout components that JDeveloper offers contain facets
- Your template must contain at least one facet

When you use a layout component from the **Component Palette**, you will see facets that this component offers in both the **Structure Panel** and the main window on the **Design** tab:



In the example above, a **Panel Splitter** component has been dropped on a page template. As you can see from the **Structure Panel**, the splitter contains two facets called `first` and `second`. In the second facet, another layout component (a `Panel Stretch Layout`) has been dropped. This is a more complex component showing five facets where you can place content (`top`, `start`, `center`, `end`, `bottom`).

When you are building a template, you will see a tab called **Facet Definitions** at the bottom of the **Create JSF Page Template** dialog:



Create JSF Page Template

Creates a new JSF Page Template, and allows you to specify facet references as well as model and view parameters.

File Name: XdmPageTemplate

Directory: C:\JDeveloper\mywork\XdmCommonUI\CommonView\public_html

Page Template Name: XdmPageTemplate

☐ Use a Quick Start Layout — One Column (Stretched)

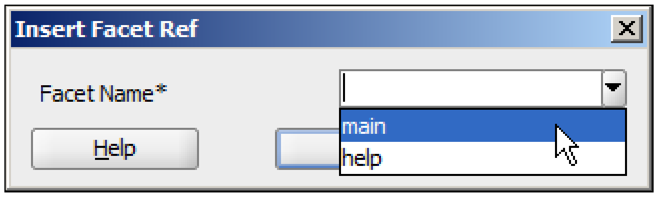
☐ Create Associated ADFm Page Definition

Facet Definitions | Attributes | Model Parameters

Name	Description
main	Page content goes here
help	Page-specific help goes here

By clicking on the green plus sign, you can define the facets you wish to make available to the programmer using the template. In the example above, the template developer decided to provide two places for content on the template: One called **main** and another called **help**.

When you start building your page template, you decide where the programmer using the template can place content. You do this by dropping a `FacetRef` component (from the **Common Components** section of the Component Palette) in your layout in the place where you want the page-specific content. When you drop a `FacetRef` component, you are prompted to choose one of the facets you defined for the template:



All the other components you place on the template page (logos, headers, and so on) cannot be changed by the programmer using the template. He can only drop his content in the facets you, as the template developer, have provided.

Attributes

When defining the template, you can also define **template attributes**. These are like parameters for the template—each page that uses the template can set different values. This is normally used in two ways:

- To place page-specific information in template areas
- To create different layouts based on the same template

Remember that you can only place your own components inside the defined facets when building a page based on a template. If you want to display, for example, the name of the current page as part of a page header (which is not a facet, but defined in the template), you use an attribute. You then place an `OutputText` or other component in the header and set its value to the attribute value using expression language. If, for example, your template attribute has the name **pageTitle**, the format of the reference is `#{attrs.pageTitle}`.

The other use for attributes is to define the parameters for the layout components you use in the template. For example, you might use expression language to define whether a specific accordion component is collapsed or expanded, the position of a splitter or similar. To achieve this, you use an expression language reference as above to set a property value for a layout component.

Using page templates

When creating a new JSF page, simply select the page template from the drop-down list.

To set attributes, you need to select the template reference on the page and use the property palette. This is not possible from the **Design** view – change to **Source** view and select the `<af:pageTemplate>` tag or use the Structure panel to find the `af:pageTemplate` element.

Framework extension classes

In addition to page flow templates and page templates that are used for the frontend part of the application, you also need another kind of template for the backend business components.

The ADF Framework uses eight base classes for Business Components. The four most important ones are:

- **EntityImpl** corresponds to one row of data and contains methods to create a row, set attribute values and perform database operations.
- **ViewObjectImpl** corresponds to the view object query. The methods in this class allow you to set bind variable, modify the WHERE clause and execute the query.
- **ViewRowImpl** corresponds to the result set from a query and contains a collection of pointers to the actual data stored in EntityImpl objects.
- **ApplicationModuleImpl** is an instance of an application module; different users will have different instances. Methods in this class handle database transactions.

The ADF framework uses four additional classes internally:

- **EntityCache** implements the cache for data retrieved from the database
- **EntityDefImpl** is the factory for producing entity objects of a specific type
- **ViewDefImpl** holds the definition of a view object
- **ApplicationModuleDefImpl** holds the definition of an application module

These classes allow you to change the way ADF works at the fundamental level – you should not want to change this.

This section will show how some of these classes are used and explain why you should create your own framework classes, and how to do that. Complete documentation of these classes can be found in the Javadoc that you can access from JDeveloper.

How Java classes are used in ADF

You saw in *Chapter 1*, *The ADF Proof of Concept*, that the business components that are central to your enterprise application are implemented in XML, with an optional Java component. If you do not define your own Java component, the framework will simply create an instance of the standard Oracle-supplied Java class. This Java class then reads the XML definition of the object you have defined and does all necessary work.

Every time you choose to generate a Java class for one of your business components (Entity Objects, View Objects or Application Modules), these will be based on the Oracle-supplied classes by default. If you look at the Java code you generate for example for an Entity Object, part of what you see is something like this:

```
…
import oracle.jbo.server.EntityImpl
…
Public class TasksImpl extends EntityImpl {
…
  public void remove() {
    super.remove();
  }
…
}
```

The keyword `extends` means that your TasksImpl class (implementing the Tasks entity object) is based on the standard `EntityImpl` object, that is, the EntityImpl class is the **superclass** of TasksImpl. The import statement at the top shows you the full path to this object: `oracle.jbo.server.EntityImpl`.

You will notice that some of the methods in this class contain a call to the corresponding method in the superclass, for example `super.remove()` in the `remove()` method.

Some Java required

Have you ever seen an implementation of large enterprise applications like Oracle E-Business Suite or SAP Business Suite that was *not* customized? Me neither.

Given the cost of customization and the pain involved when upgrading a heavily customized "standard" solution, why do organizations still do it?

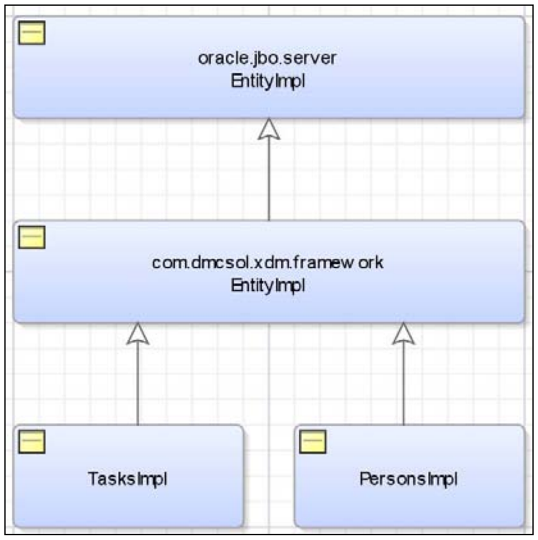
Because it is impossible to build a standard system that will meet every need the organization has. Similarly, it is impossible for Oracle Application Development Framework to directly meet every requirement in your enterprise application. That is why we have the option to create Java classes: To implement the specific functionality that the framework does not have built in.

The place for framework extension classes

By default, when you generate a Java class for a business component, it will directly extend a class from the `oracle.jbo.server` package. You do not want this, because you might not always be happy with the way the ADF framework does things.

If you want to change the way data is stored in a single entity object, you can just change that one Java class. But if you are making the same kind of change to multiple objects, good programming practice dictates that you should make this change only once. This might be the case if you want to make all entity objects call a PL/SQL API package instead of accessing the table directly. In an object-oriented language, this change should be made in the superclass that your objects inherit from. However, if you inherit directly from the Oracle-supplied classes, you cannot change the superclass without invalidating all claims to support from Oracle.

That is why you need to place your own framework extension classes between the Oracle-supplied classes and your own implementations, as shown next:



In this way, all of your own implementation classes inherit from a class under your own control. For example, your `TasksImpl` class will extend your own `com.dmcsol.xdm.framework.EntityImpl` class.

Multi-layer framework extension

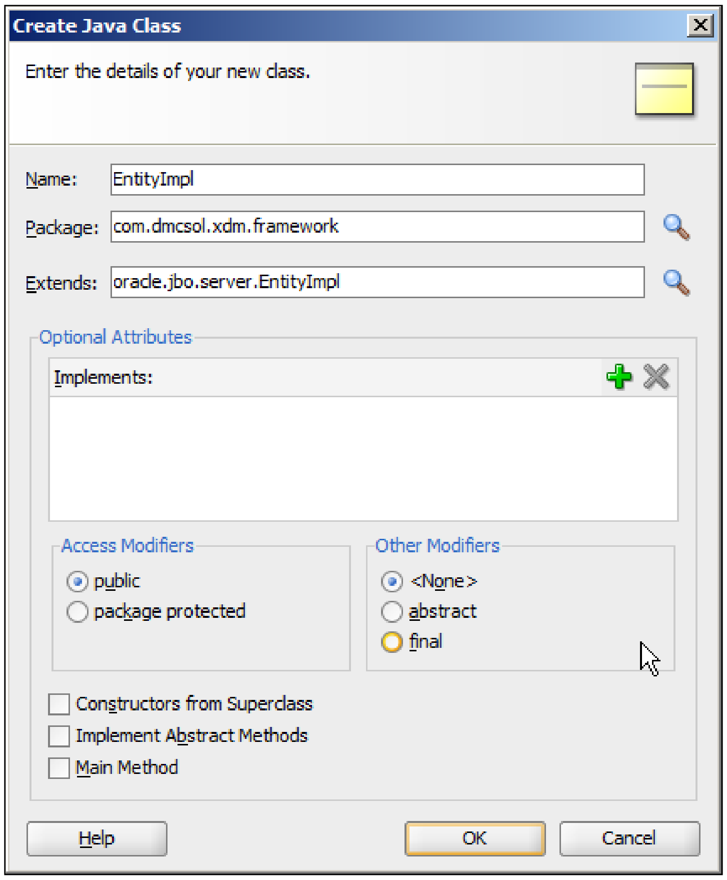
If you already know that your organization will be creating dozens of Oracle ADF applications, you might want to think about creating two layers of framework extension classes: One general (for all your ADF applications) that extends the Oracle-supplied classes and then specific framework extensions for each project, extending your own general extension class.

Creating framework extension classes

The concept of framework extensions illustrates the beauty of object-oriented programming: You simply have to create an empty class extending the Oracle-supplied class and do nothing else. All of the methods from the Oracle-supplied superclass are automatically available and will be executed at run time. As you decide you need to implement specific functionality in your framework classes, you simply add the relevant methods and they will automatically override the method from the superclass.

If you use the recommended workspace structure, your framework extension classes should go into your CommonCode workspace. If you have not already created this, do this now as described above for the CommonUI workspace. In Step 2 of the wizard, call your project **FrameworkExtension** and choose to include **ADF Business Components** technology. In Step 3 of the wizard, set the **Default Package** to the sub-package `framework` under your project base package (for example, `com.dmcsol.xdm.framework`).

In this project, choose **File | New** and then **General | Java | Java class**. Give your class the same name as the Oracle class you extend (for example, `EntityImpl`) and place it in your framework package (`<company base>.<project>.framework`, for example, `com.dmcsol.xdm.framework`). Extend the relevant Oracle class (for example, `oracle.jbo.server.EntityImpl`). Leave **Access Modifiers** at **public** and uncheck the checkboxes:


Create Java Class

Enter the details of your new class.

Name: EntityImpl
Package: com.dmcsol.xdm.framework
Extends: oracle.jbo.server.EntityImpl

Optional Attributes

Implements:

Access Modifiers: [✓ public] package protected

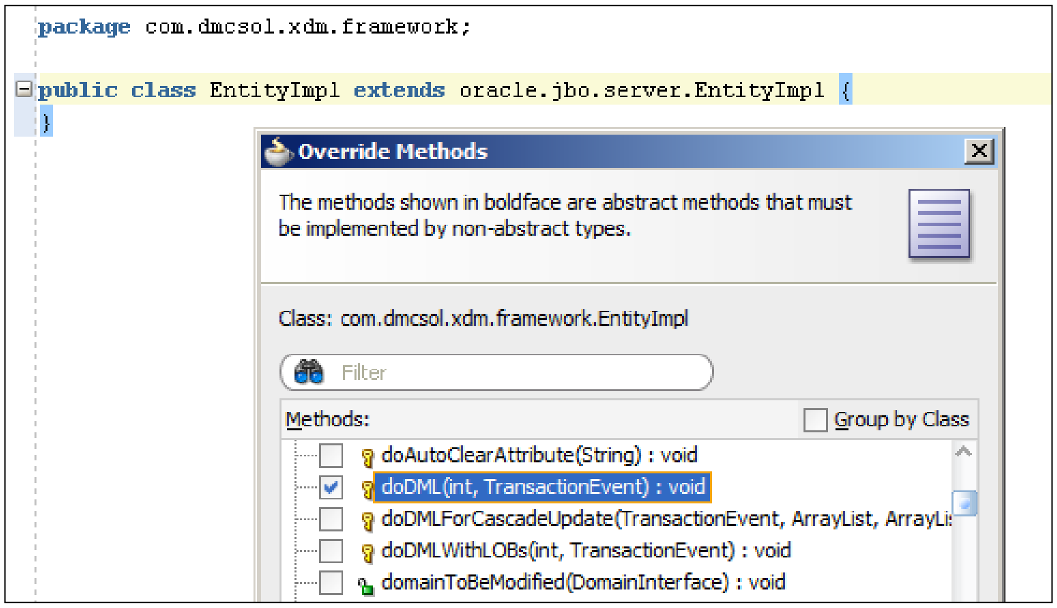
Other Modifiers: [✓ <None>] abstract final

☐ Constructors from Superclass
☐ Implement Abstract Methods
☐ Main Method

Then click **OK** to create the class. It will be really simple – something like this:

```
package com.dmcsol.xdm.framework;

public class EntityImpl extends oracle.jbo.server.EntityImpl {
}
```

You do not have to put any content into this class right now. If you later find that you need to override a method (for example the `doDML()` method that is called every time ADF issues an INSERT, UPDATE or DELETE to the database), right-click inside the class and from the context menu choose **Source | Override Methods**:



```
package com.dmcsol.xdm.framework;

public class EntityImpl extends oracle.jbo.server.EntityImpl {
}
```

Override Methods

The methods shown in boldface are abstract methods that must be implemented by non-abstract types.

Class: com.dmcsol.xdm.framework.EntityImpl

- [ ] doAutoClearAttribute(String) : void
- [✓] doDML(int, TransactionEvent) : void
- [ ] doDMLForCascadeUpdate(TransactionEvent, ArrayList, ArrayLi
- [ ] doDMLWithLOBs(int, TransactionEvent) : void
- [ ] domainToBeModified(DomainInterface) : void

From the **Override Methods** dialog, you can check the checkboxes for the method you want to override.

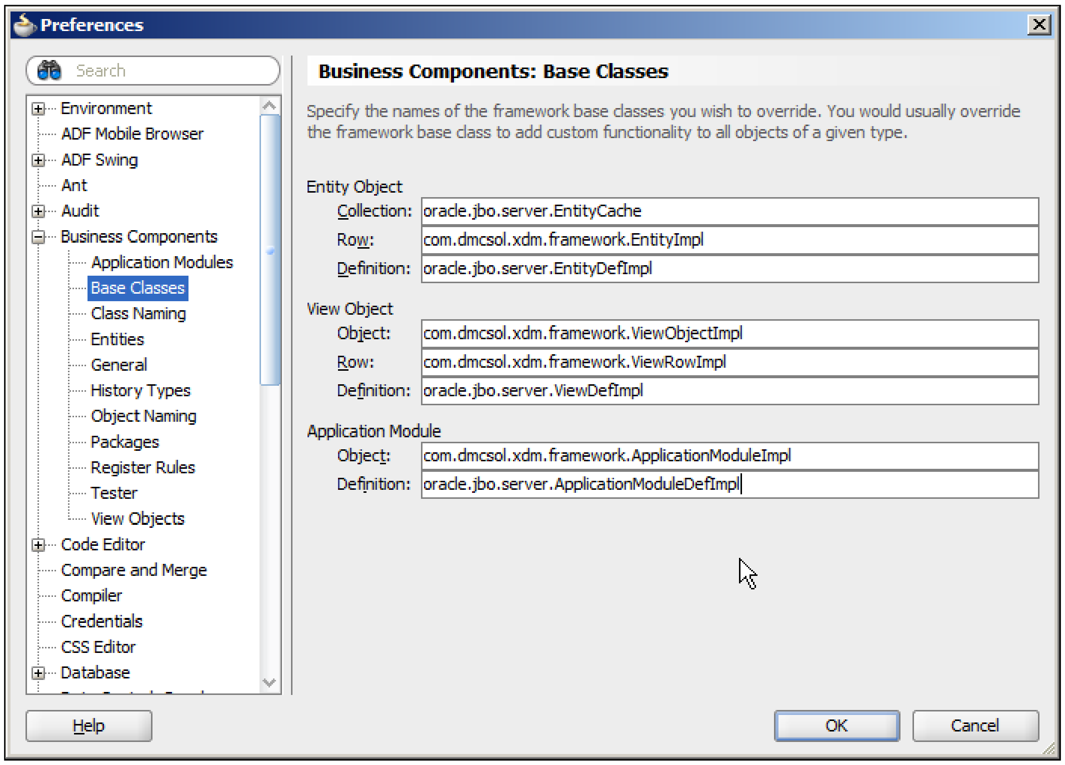
Repeat the above procedure for the remaining three ADF Business Component base classes, so you have your own version of:

- `EntityImpl`
- `ViewObjectImpl`
- `ViewRowImpl`
- `ApplicationModuleImpl`

Do not override `EntityCache`, `EntityDefImpl`, `ViewDefImpl`, and `ApplicationModuleDefImpl` unless you are very familiar with ADF and are sure you understand the implications of changing them.

Using framework extension classes

Once you have created your own version of the four important classes, you must set up JDeveloper to use your classes instead of the Oracle-supplied classes. It is easiest to configure JDeveloper to use your own classes for all projects by choosing **Tools | Preferences** and then **Business Components | Base Classes**. In the dialog box, fill in the name of your own classes for the four important base classes, like this:



Business Components: Base Classes

Specify the names of the framework base classes you wish to override. You would usually override the framework base class to add custom functionality to all objects of a given type.

Entity Object
Collection: oracle.jbo.server.EntityCache
Row: com.dmcsol.xdm.framework.EntityImpl
Definition: oracle.jbo.server.EntityDefImpl

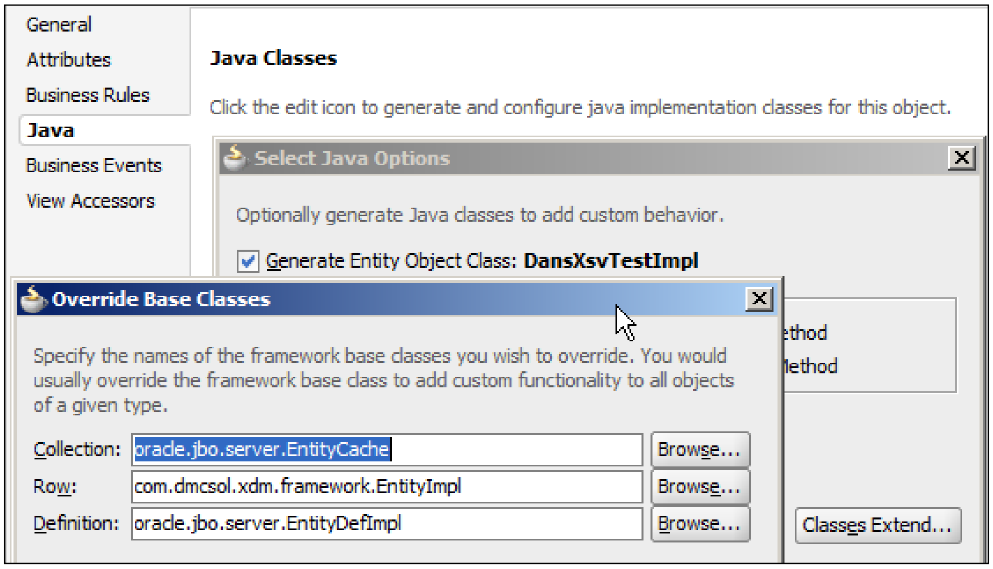
View Object
Object: com.dmcsol.xdm.framework.ViewObjectImpl
Row: com.dmcsol.xdm.framework.ViewRowImpl
Definition: oracle.jbo.server.ViewDefImpl

Application Module
Object: com.dmcsol.xdm.framework.ApplicationModuleImpl
Definition: oracle.jbo.server.ApplicationModuleDefImpl

This will cause JDeveloper to always use your framework extension classes whenever you generate Java classes for an ADF business component.

If you are working on multiple projects, some of which use different framework extension classes, you can also define the base classes at the project level for your model project (under **Project | Properties**).

If you have an existing business component that you want to "re-fit" from the standard Oracle class to your own framework extension class, you can choose the **Java** sub-tab for the component and click on the **Classes Extend** button to set the base class for your business component:



Click the edit icon to generate and configure java implementation classes for this object.

Optionally generate Java classes to add custom behavior.

☑ Generate Entity Object Class: **DansXsvTestImpl**

Specify the names of the framework base classes you wish to override. You would usually override the framework base class to add custom functionality to all objects of a given type.

Collection: oracle.jbo.server.EntityCache
Row: com.dmcsol.xdm.framework.EntityImpl
Definition: oracle.jbo.server.EntityDefImpl

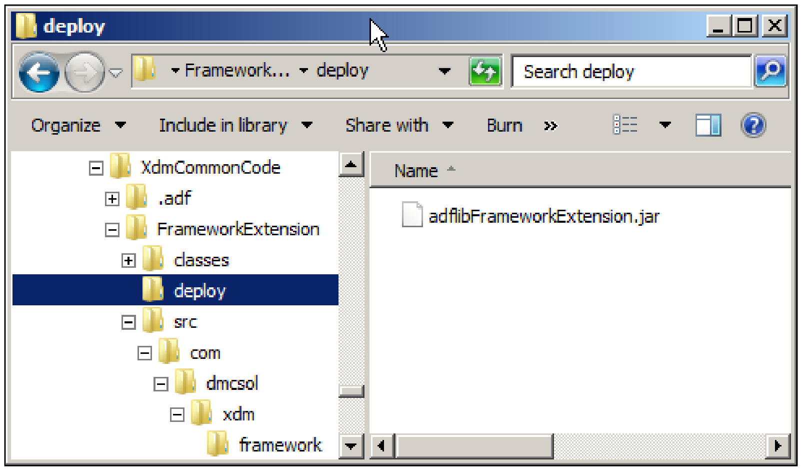
Packaging your Common Code

Both your CommonUI and your CommonCode workspaces need to be packaged into ADF Libraries so they can be made available to the developers working on the individual subsystems your enterprise application will consist of.

To deploy a project to an ADF Library, select the project and choose **File | New**, choose **Deployment Profiles** (under **General**) and select **ADF Library JAR File**. Give your ADF Library a name (use the project name, prefixed with `adflib`, for example, `adflibFrameworkExtension`) and click **OK**. You do not need to make any selections in the **Edit ADF Library JAR Deployment Profile Properties** dialog—simply click **OK** to finish creating your deployment profile.

Once you have a deployment profile, you can right-click on your project, choose **Deploy** and then the name of your deployment profile. You only need to click **Next** and then **Finish** to perform the actual deployment that creates the ADF library file.

The ADF library file is created in a new directory called `deploy` that will be created under your project (for example `C:\JDeveloper\mywork\XdmCommonCode\FrameworkExtension\deploy`). The file name is the same as the name of your deployment profile (for example, `adflibFrameworkExtension.jar`):



Keeping your ADF libraries under control

It is a good idea to copy your ADF libraries from the deploy directories of each project to one common library directory, and import them into your version control system there. In this way, the latest approved and tested libraries are available to all developers on the project, while the team working on the common workspaces are free to build, test, and commit code without disrupting work on other projects.

Summary

This was the final piece of preparation for building your enterprise application. Now you have the templates and framework classes you need, all packaged nicely in ADF Libraries ready for use. In the next chapter, we will test that it all works by re-building the XDM proof of concept functionality using all the correct enterprise methods, tools and templates.

6
Building the Enterprise Application

Finally! We are back to real programming! With the infrastructure in place and our tools set up correctly, we are ready to start building our real enterprise application. In this chapter, you will be implementing two subsystems containing the two use cases we prototyped in the proof of concept in *Chapter 1*, *The ADF Proof of Concept*, and will be collecting them into a completed enterprise application.

Decorate the project room

The room where your team is working should be decorated. Not with Christmas ornaments, but with something infinitely more useful: The Data Model. Even if you have not used JDeveloper to define your data model, you can use JDeveloper to create a database diagram.

Print out the entire data model for your system in a size large enough to be able to read every column in every table. Unless you work in the construction industry, you probably do not have a printer large enough to fit everything on one sheet—go to your local print shop or get out the scissors and scotch tape.

Structuring your code

Your enterprise application is going to consist of hundreds of objects—entity objects, view objects, application modules, task flows, page fragments, and many others. That makes it imperative that you keep everything in a logical structure. A good structure also allows you to partition work between the many people who will be working in your team, and ensures that everyone can find what they need.

Workspaces

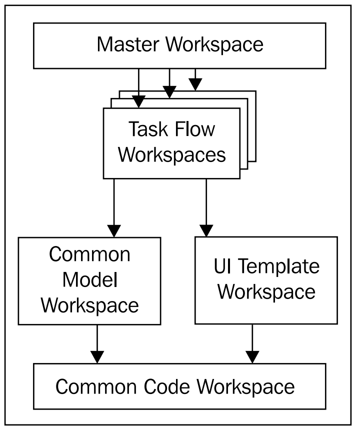
Remember from *Chapter 3*, *Getting Organized*, that we divide up the application between a number of workspaces. In recent versions, JDeveloper has unfortunately started using the word "application" for what used to be called **workspace**. While it is correct that you need a workspace to build an application, the opposite is not true – you will have many workspaces that are not applications.

[If you look at the definition created in the file system, you will find that the extension for the workspace definition file is actually still `.jws` (which used to mean Java WorkSpace). So whenever you see JDeveloper use the word "Application", think "workspace".]

There is no need to try to keep your entire enterprise application in one JDeveloper "application". Using ADF Libraries, it is easy to combine separate workspaces, and having a number of smaller development teams working on separate subsystems ensures proper modularity of your enterprise application.

The workspace hierarchy

Your workspaces will be arranged in a hierarchy as described in *Chapter 3*, *Getting Organized*, and the individual workspaces at lower levels will use the workspaces at higher levels:



You already built the **Common Code Workspace** in *Chapter 5*, *Prepare to Build*, – this workspace contains the Framework Extension Classes and will eventually contain all the other general utility classes you build during the course of the project.

You have also already built the **Common UI Workspace** in *Chapter 5*, *Prepare to Build*, – this workspace contains your page flow and page templates. When we start customizing the application appearance by developing a custom "skin" in *Chapter 8*, *Look and Feel*, the style sheet and other UI artifacts from this process will also go into the Common UI workspace.

The **Common Model Workspace** is new, and you will start building this workspace later in this chapter. This is where all your entity objects go, as well as any common view objects that will be shared across the whole application – for example, all the view objects used by your value lists.

Creating a workspace

Whenever you need a new subsystem workspace, you choose **File | New**, **General**, **Applications, Fusion Web Application**.

Name the workspace in accordance with your naming standards. If your company domain name is **dmcsol.com** and your project abbreviation is **xdm** ("neXt generation Destination Management"), all your project code should be placed in sub-packages under `com.dmcsol.xdm`. Code for your individual task flows go into sub-packages named after the subsystem the task flow implements. For example, you might consider the Task Overview and Edit use case (UC008) a subsystem and place all code for this in a package called `com.dmcsol.xdm.uc008`. Refer to *Chapter 3*, *Getting Organized*, for more on package naming.

JDeveloper will automatically create two projects inside your workspace called `Model` and `ViewController`. Because these project names are used in various configuration files, you should change the project names to reflect the subsystem you are implementing, for example to `uc008Model` and `uc008View`.

The model project gets `.model` added to the package name and the view project gets `.view` added for a total package name like `com.dmcsol.xdm.uc008.view`. There is no need to change these sub-package names.

Separate names for separate tasks

It is important that you use separate package and project names for separate task flows. If you accidentally use the same name in two different task flows, you will experience mysterious errors once you combine them into enterprise application.

Working with ADF Libraries

ADF Libraries are built from projects inside workspaces, so each workspace will produce one or more ADF Libraries for other workspaces to use. This section suggests a procedure for working productively with ADF Libraries, but if you have another method that works, that is fine too.

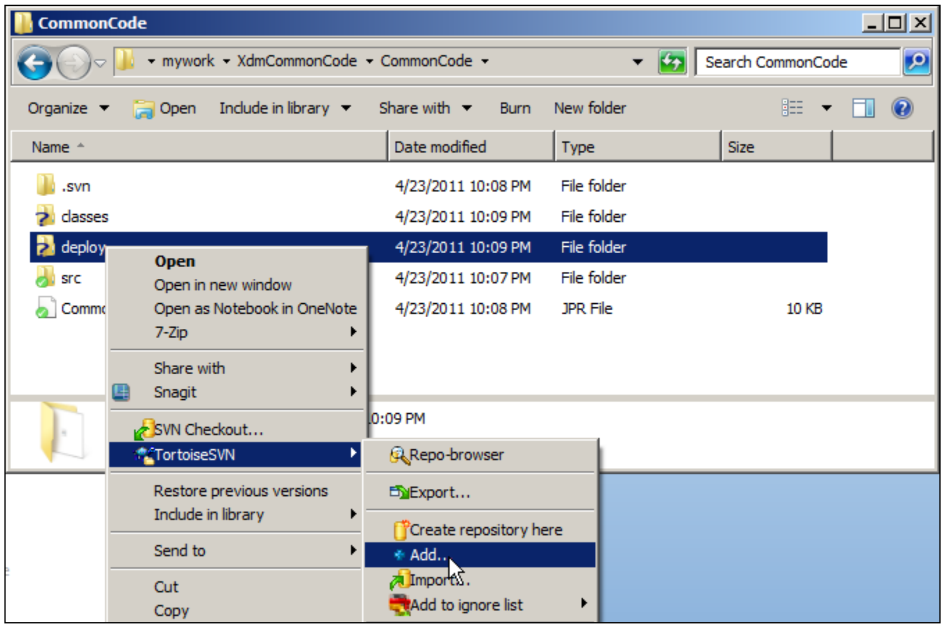
Version Control Outside JDeveloper

Because the ADF libraries you produce do not show up in the Application Navigator in JDeveloper, it is easiest to version control them outside JDeveloper. If you are using Subversion, TortoiseSVN is a good client that integrates with Windows Explorer for right-click version control.

ADF Library workflow

The ADF Library workflow goes like this:

- The developer builds components in a workspace and performs his own testing.
- When the developer is satisfied, he creates an ADF Library from within JDeveloper. He then goes outside JDeveloper, navigates to the **deploy** folder inside his project and adds it to the source code repository:

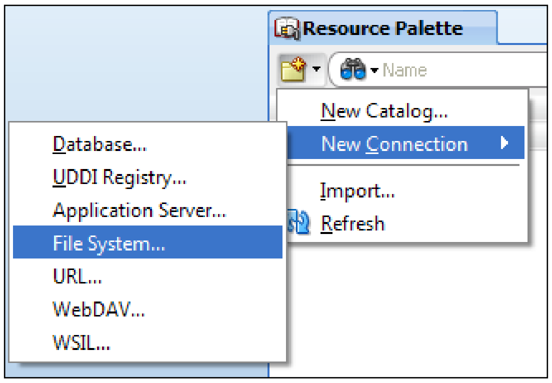

CommonCode
mywork ▸ XdmCommonCode ▸ CommonCode
Search CommonCode
Organize Open Include in library Share with Burn New folder
Name Date modified Type Size
.svn 4/23/2011 10:08 PM File folder
classes 4/23/2011 10:09 PM File folder
deploy 4/23/2011 10:09 PM File folder
src 4/23/2011 10:07 PM File folder
Comm 4/23/2011 10:08 PM JPR File 10 KB
Open
Open in new window
Open as Notebook in OneNote
7-Zip
Share with
Snagit
SVN Checkout...
TortoiseSVN
Restore previous versions
Include in library
Send to
Cut
Copy
Repo-browser
Export...
Create repository here
Add...
Import...
Add to ignore list

- The Build/Deployment Manager gets the entire workspace from the source code repository and copies the ADF Library to a common ADF Library folder in his local file system.
- The Build/Development Manager works with the test team to test the ADF Library file. When satisfied, he commits the common library folder to the source code repository.
- Developers check out the common library folder from the source code repository to a directory on their local file system (for example, `C:\JDeveloper\XdmLib`).
- Developers create a file system connection to the common library folder on their local machine as described next.

Using ADF Libraries

Because ADF Libraries are simply JAR files, it is possible to just add them to your JDeveloper projects by choosing **Project Properties**, **Libraries**, **Add JAR/Directory**. If you use this method, you must add the individual JAR files, not just the directory.

However, you get a better overview of your ADF Libraries if you create a file system connection to the directory containing your JAR files. To do this, open the **Resource Palette** from the **View** menu. Click the **New** button in the **Resource Palette** and choose **New Connection**, **File System**:



Give your connection a name (for the sample project in this book, `XdmLib`) and point to your library directory (for example, `C:\JDeveloper\XdmLib`).

Now the **Resource Palette** will show you the available libraries from all workspaces. To actually use a library in a project, you simply select a project in the **Application Navigator** and right-click on individual library in the **Resource Palette** and choose **Add to Project** or drag it from the **Resource Palette** onto the project.

Building the Common Model

As discussed above, your enterprise ADF application should be based on three common workspaces: Common Code, Common UI, and Common Model. You created the first version of the Common Code and Common UI workspaces in *Chapter 4, Productive Teamwork*– all that is missing now is the Common Model workspace.

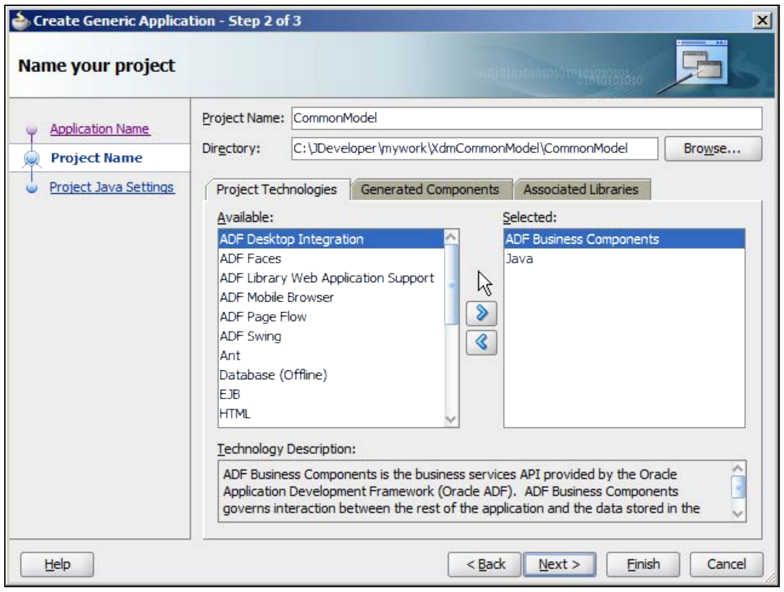
In this workspace, you will create all the entity objects used in the entire application, as well as the view objects used for common lists of values used throughout the application. The common model workspace can also include other view objects, if you can identify view objects that will be used in several places in the application.

Specialized view objects for each screen will go into the subsystem workspaces that we will build later in this chapter.

Creating the workspace

Create the workspace using **File | New** and **Generic Application**. Provide workspace and package names in accordance with your naming standard—for the XDM project of DMC Solutions, the workspace name is `XdmCommonModel` and the package is `com.dmcsol.xdm.model`.

In Step 2 of the wizard, give the project name **CommonModel** and select **ADF Business Components** technology. Because ADF Business Components depends on Java technology, this technology will automatically be added as well:


Create Generic Application - Step 2 of 3

Name your project

Application Name
Project Name
Project Java Settings

Project Name: CommonModel
Directory: C:\JDeveloper\mywork\XdmCommonModel\CommonModel

Project Technologies | Generated Components | Associated Libraries

Available:
ADF Desktop Integration
ADF Faces
ADF Library Web Application Support
ADF Mobile Browser
ADF Page Flow
ADF Swing
Ant
Database (Offline)
EJB
HTML

Selected:
ADF Business Components
Java

Technology Description:
ADF Business Components is the business services API provided by the Oracle Application Development Framework (Oracle ADF). ADF Business Components governs interaction between the rest of the application and the data stored in the

Using framework extension classes

If you did not set up JDeveloper to use your own framework extension classes when you built them in *Chapter 5*, *Prepare to Build*, please do so now. Go to **Tools | Preferences** and select **Business Components**, **Base Classes**. For the four types of objects where you created your own classes (EntityImpl, ViewObjectImpl, ViewRowImp, ApplicationModuleImpl), change from `oracle.jbo.server` to the package name of your own classes (for example, `com.dmcsol.xdm.framework`). This tells JDeveloper that every ADF Business Object built from now on should be based on your own classes.

Then open the **Resource Palette** and find your shared ADF libraries. Right-click on the Common Code library (for example, `adflibXdmCommonCode`) and choose **Add to Project**.

Entity objects

In an ADF application, you normally have one entity object for every table in your application. Since these entity objects can be re-used by many different view objects, it makes sense to develop them once in the Common Model workspace and then deploy them to all other parts of the application in the form of an ADF Library.

You can define validation rules on your entity objects as well. Validation placed here will always be executed, no matter which view object uses the data.

Validation in the database

Remember that the only way to make sure a validation rule is *always* executed is to place it in the database. Critical data validation logic must be implemented in the database as well. Validation at the entity object level will only be executed when data is changed through the ADF application.

If your database contains foreign keys defining the relationship between tables, the "Business Components from Tables" wizard in JDeveloper will build all of your entity objects and associations. If you do not have foreign keys in the database, the wizard can only create the entity objects – you will have to add all associations by hand (choose **File | New** and then **Business Tier**, **ADF Business Components**, **Association**).

Because entity objects cannot be tested in isolation, you should also generate default view objects (one for each entity object), and one application module. This allows you to test your entity objects through the Business Components Tester built into JDeveloper. Your default view objects should be named …`DefaultVO` to make clear to everyone that they are not intended for use in the application; similarly, your test application module should be named something like `CommonModelTestService`.

To create your entity objects, default view objects, and application module, perform the following steps:

1. Choose **Tools | Preferences** and then **Business Components**, **Object Naming**. Set the suffix for object **View Object** to **DefaultVO**. This makes JDeveloper build view objects ending with ...`DefaultVO` as we wish.
2. In the `CommonModel` project, choose **File | New** and then **Business Tier**, **ADF Business Components**, **Business Components from Tables**.
3. Create a connection to the database where your tables exist and in Step 1 of the wizard click the **Query** button to find all your tables. Move them to the **Selected** box and click **Next**.
4. In Step 2 of the wizard, move all the entity objects to the **Selected** column. This generates default view objects for all your entity objects. In Step 3, just click **Next**.
5. In Step 4, *deselect* the checkbox **Application Module**. We do not want the default application module from this wizard, because it includes our view objects in every conceivable combination, and that is unnecessary here. Then simply click **Finish** to close the wizard.
6. In the application navigator panel, expand the **model** node, then the **view** node. Select the **link** node, press *Delete* and confirm the deletion. This removes all the unnecessary view links that the wizard built for us.
7. Choose **File | New** and then **Business Tier**, **ADF Business Components**, **Application Module**. Name the application module **XdmCommonModelTestService** and click **Next**. In step two, expand the tree in the left-hand box and add all the ...`DefaultVO` view objects. Then click **Finish** to close the wizard.

Do not mix test and production

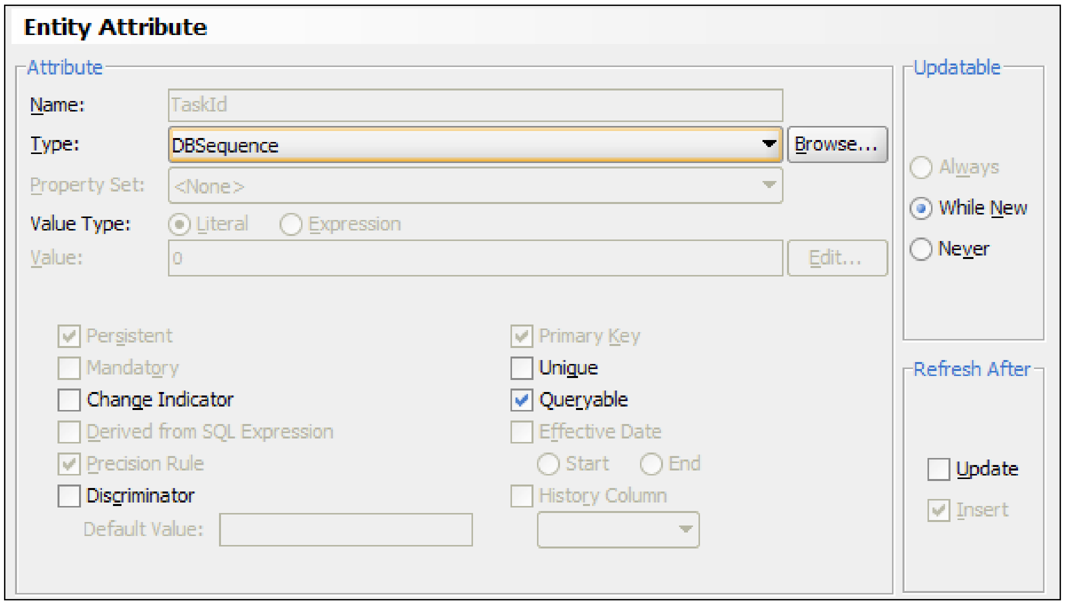
You need an application module in order to test your common model entity objects. There is no way of preventing this application module from showing up in the Data Control panel when using the common model library in your subsystem workspaces – so it should have a name that makes clear to all developers that it is not intended for production use.

8. Choose **Tools | Preferences** and then **Business Components**, **Object Naming**. Set the suffix for object **View Object** to **VO**.

This procedure creates one entity object for every table, one default view object for every entity object (simply containing all the attributes in the entity object) and one application module containing all your default view objects. The purpose of the default view objects and the application module is only to allow you to test the entity objects using the business components tester – these default view objects should not be used in the production application.

Generating primary keys

It is a good idea to let a database trigger create the primary key value for all tables with numeric keys – the sample XDM database script contains sequences and triggers that do this. When using trigger-supplied ID values like this, you need to tell the ADF Framework to expect that the value for the ID columns will be changed when a record is created in the database. To do this, open each entity object, choose the **Attributes** sub-tab on the left, double-click the ID attribute and set the **Type** to **DBSequence**:



Entity Attribute

Attribute

Name: TaskId

Type: DBSequence

Property Set: <None>

Value Type: [✓ Literal] Expression

Value: 0

[✓] Persistent
[ ] Mandatory
[ ] Change Indicator
[ ] Derived from SQL Expression
[✓] Precision Rule
[ ] Discriminator
Default Value:

[✓] Primary Key
[ ] Unique
[✓] Queryable
[ ] Effective Date
Start End
[ ] History Column

Updatable: Always [✓ While New] Never

Refresh After:
[ ] Update
[✓] Insert

Business rules

The above procedure creates simple entity objects containing only a few simple business rules derived from the database (whether an attribute is mandatory and a length limitation based on the database column definition).

The common model team should add the additional business rules your application calls for to the entity objects in the common model workspace. This is done on the **Overview** tab under the **Business Rules** sub-tab in each entity object.

Handling PL/SQL business rules

The business rules that you might have implemented as CHECK constraints in the database are not automatically picked up by JDeveloper – the tool has no way of translating PL/SQL database logic into ADF business rules. Functionally, it is enough to do the validation in the database, but you can provide the user with better error messages if you also implement the most important business rules in ADF.

User interface strings

You should also define the default labels used for the attributes of your entity objects here in the Common Model. This is done by selecting an attribute, clicking the **Edit Selected Attribute** button (the little pencil icon) and going to the **Control Hints** sub-tab. The **Label Text**, **Tooltip Text** and other control hints you define here will become the default wherever the attribute is used.

The user interface strings defined under **Control Hints** are stored in a resource bundle – refer to *Appendix A*, *Internationalization*, for more information about resource bundles and how to use them to translate your application.

Common view objects

Your application is almost certain to contain some view objects that can be shared among different task flows. The view objects used for drop-down list boxes and groups of radio buttons fall into this category; there might also be other common data objects used throughout the application that it makes sense to place in the Common Model workspace.

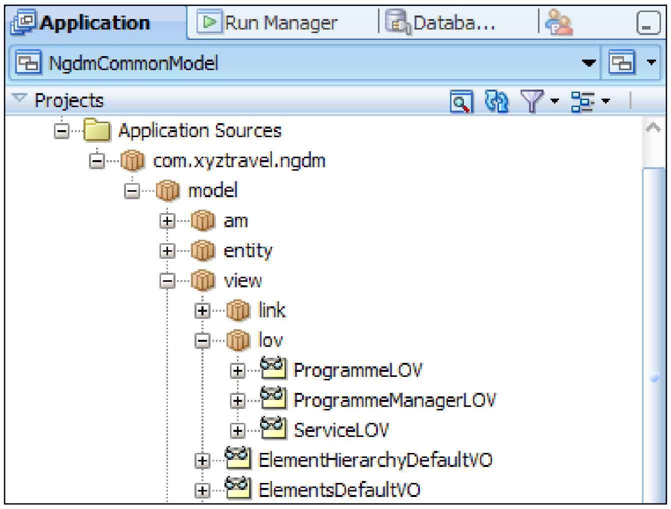
When all the entity objects and default view objects have been built, the team in charge of the Common Model must look through all the UI sketches for the entire application to determine which value list view objects are necessary.

For the example of this chapter, we will only consider two user cases: UC008 Task Overview and Edit and UC104 Person task timeline. You can find the user interface sketches further along in this chapter. Looking at these sketches, the common model team decides they need to implement three value list view objects:

- ServiceLOV
- ProgrammeLOV
- ProgrammeManagerLOV

The procedure for creating these is:

1. In the **CommonModel** project of the **XdmCommonModel** workspace, create a new view object called **ServiceLOV** (use the suffix `LOV` for view objects intended for lists of values). Place it in a sub-package `lov` under `view` to keep the LOVs separate from the other view objects:



2. Choose to base it on the **Elements** entity object and *deselect* the checkbox **Updatable**.
3. Choose the **ElemKey** and **Description** attributes. (An LOV view object should include the ID or key attribute as well as any identifying attributes that the user of the LOV view object might want to display.)
4. In Step 5, add an **Order By** clause.
5. Click **Finish**.
6. Repeat steps 1-7 for other requested view objects (**ProgrammeLOV**, **ProgrammeManagerLOV**). **ProgrammeManagerLOV** would be based on `Persons` but with a **Where** criteria limiting the LOV view object to those persons where `PROGRAMME_MANAGER_YN = 'Y'`.
7. Create a new application module called **XdmLovService**. In the Data Model step of the wizard, add the three …`LOV` view objects.

Testing the Common Model

You would normally not create test cases for entity objects that simply perform create, read, update and delete operations and implement the default business rules. But if you have defined additional business rules, you need to test these.

Depending on your testing strategy, you might decide to write unit tests to check your validation rules programmatically, or you might check them manually. Creating default view objects and an `XdmCommonModelTestService` application module allows you to run the application module in the Business Components tester. Here, you can change data to verify any validation you have added to your entity objects.

Similarly, you need to test the value list view objects you created and added to the ...`LovService` application module, either with unit test cases or manually. We will discuss testing strategies in more detail in *Chapter 7*, *Testing your Application*.

Exporting an ADF Library

Once you have defined and tested all the entity objects and view objects you need, you need to deploy them to an ADF Library that can be included in other workspaces. To do this, perform the following steps:

1. Choose the **CommonModel** project in your **XdmCommonModel** workspace.
2. Choose **File | New** and then **Deployment Profiles** (under **General**). Choose **ADF Library JAR File**. Give your deployment profile a name that includes the prefix `adflib`, your project abbreviation and "CommonModel", for example, **afdlibXdmCommonModel**. This will be the name of the JAR file this deployment profile will build.
3. Click **OK** a couple of times to close the wizard.
4. Right-click on your **Model** project and choose **Deploy**, **XdmCommonModel**. Click **Finish** to generate your ADF library JAR file.

Now your common model has been packaged up into a JAR file. You can find it in the deploy subdirectory of your project. If you point to the **CommonModel** project without clicking, you will see a pop-up telling you the location of the project definition file. The directory part of this will be something like `C:\JDeveloper\mywork\XdmCommonModel\CommonModel`. If you go to that directory, you will find a `deploy` subdirectory, and inside that your JAR file.

As described in the earlier section on working with ADF Libraries, you must manually add this deploy directory to your source control system so that your Build/Deployment manager can pick it up, have it tested and distributed.

For the purposes of the example in this chapter, you can take the role of Build/Deployment manager and copy your `afdlibXdmCommonModel.jar` file from `C:\JDeveloper\mywork\XdmCommonModel\CommonModel\deploy` to `C:\JDeveloper\XdmLib`.

Organizing the work

A climber on an expedition to Mount Everest does not just step out of his tent one morning in base camp, pick up his rucksack, and head for the mountain. He starts out by carefully planning the stages of his climb, splitting the total task into smaller subtasks. This allows him to focus on the task at hand and give his full concentration to climbing through the dangerous Khumbu Icefall at 18,000 feet – without worrying about the Hillary Step at 28,840 feet just below the summit.

While enterprise ADF development might lack the glory (and danger) of climbing Mount Everest, the idea of concentrating on the one task at hand still applies. Especially while you are new to ADF development, concentrate on one task to avoid being overwhelmed by the full complexity of the whole application. This is another reason to build your enterprise application as a number of separate subsystems.

Preconditions

Just like the Mount Everest climber, you need skills, tools and favorable conditions. You have built up your skills by taking classes, reading books and blogs, doing exercises and building smaller applications. You have set up your tools as described in *Chapter 4*, *Productive Teamwork*. The last precondition is favorable conditions.

To the Himalayan climber, favorable conditions means good, stable weather. To an ADF developer, favorable conditions means good, stable user requirements. If you do not have good requirements, your odds of success decrease – just like those of a climber starting out in unsettled weather.

Good user requirements include:

- Textual description of the purpose of the user case.
- Description of all relevant business rules.
- Sketches of all screens, annotated with the requirements not obvious from the visual appearance. For example, if the UI sketch shows a drop-down list box, it must be annotated with a data source providing the values on the list.

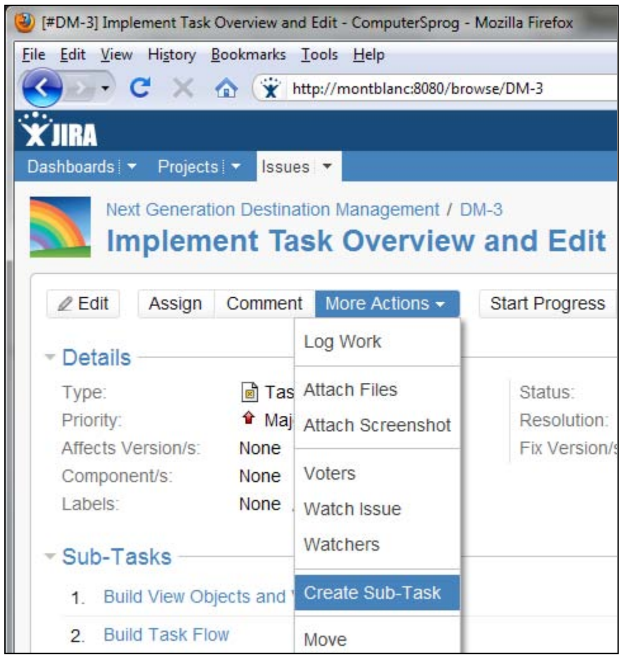
Decorate the project room

Hang all the UI sketches you have on the walls of your project room, together with any graphical mockups showing colors and common elements. As you start building real screens, print these out and hang on the wall as well. Having these visual elements in front of everyone improve the consistency of your user interface.

Development tasks

If you are using a task tracking system like Jira, you are likely to be assigned a fairly coarse-grained task from the Work Breakdown Structure the project manager has constructed – something like "Implement Task Overview and Edit".

Tasks at this level of granularity must be split into useful subtasks so a developer can work on it in an efficient and effective manner. With Jira (and probably most other task tracking solutions), you can split a task into sub-tasks. If you set up Oracle Team Productivity Center as described in *Chapter 4*, *Productive Teamwork*, you will have access to some basic Jira functionality directly within JDeveloper, but at the time of writing, not the ability to create sub-tasks. For this, you need to change to the Jira web interface, bring up the task you want to split and choose **Create Sub-Task** (under the **More Actions** button in the Jira web interface):



Your new tasks get normal Jira issue numbers, so they are available for you to work on in JDeveloper. Remember that you can identify one task as the **Active Work Item** – JDeveloper will automatically suggest that you link this item to each Subversion commit, allowing you to track code commits against tasks.

You will normally need the following sub-tasks:

- Create Business Components
- Implement Business Logic
- Create Task Flows
- Review Task Flow
- Create Page Fragments
- Implement UI Logic
- Define UI test
- Review UI test

Use this list as a starting point, and add additional subtasks if your specific use case calls for them, or remove any that are not relevant.

Creating business components

Your Common Model project will already contain all the entity objects as well as view objects for the value lists. When developing the subsystem, you only need to define the view objects specific to the subsystem, and an application module.

Building view objects, view links, and application module

To determine which view objects you need, look at the user interface sketch. Each separate block of data on your user interface sketch indicates a potential view object, and any blocks of data with a master-detail connection between them indicate a view link between the blocks.

One view object can be used in multiple places – even on separate tabs or pages – if you want all data blocks to refer to the same record. For example, you only need one Customer view object even if you display the customer name on one tab and the address information for that customer on another tab. On the other hand, if you have one tab showing customer names and another tab where the user can page through customer addresses independent of the selection on the first tab, you need two view objects.

Remember that you can combine data from multiple entity objects in one view object. This means that you do not need to create a view link just to retrieve a value from another table – you only need a view link if you want to display a master/detail relationship.

In the famous SCOTT schema that has been delivered with almost all versions of the Oracle database, there is a table EMP with a foreign key pointing to the DEPT table. The EMP table contains the department number, but not the department name. If you wish to show employees together with department name, you base your view object on both the Emp and Dept entity objects, and select the necessary attributes from the Emp entity object and the department name (Dname) attribute from the Dept entity object. No view link is necessary for this.

If your user interface includes value lists, you need to determine which view object can deliver the data you need (both the code/key and the value displayed to the user). Most value lists will be based on common view objects that are used throughout the application. These should be part of the Common Model workspace that you include in your subsystem workspace.

If you find that you need a value list that is not in the common workspace, consider if this value list is really unique to the task flow you are implementing. If you are sure this is the case, go ahead and build it in your subsystem workspace. However, if you are *not* sure that you will be the only one ever using that value list, talk to the team in charge of the Common Model workspace to get your value list view object included in the common model for everyone to use.

Your view objects might also include named view criteria, bind variables, and so on.

When you are done building view objects, create an application module for your subsystem.

Implementing business logic

The team building the Common Model has already implemented some business logic as **Business Rules** in the entity objects. The remaining business logic goes into your view objects and application module. How to program ADF business logic is outside the scope of this book, but an internet search will quickly uncover many ADF programming resources on the Oracle Technology Network and elsewhere.

Remember the database

Some business logic is better implemented as stored procedures in the database. If you are a Java programmer not familiar with the capabilities of the database, remember to talk to the database developers on the team to make sure the business logic is implemented in the right place.

Testing your business components

Once your view object is complete, test it through the business component tester. Depending on your test strategy, you might also want to write JUnit test cases that verify your view criteria; we will discuss testing in detail in *Chapter 7*, *Testing Your Application*. Only write test cases for complicated stuff – you can trust the ADF framework to retrieve the data you specify.

Creating task flows

The frontend part of the application should be built with **bounded task flows** using **page fragments**.

A bounded task flow can contain screens, define transitions between them and even include calls to code or other bounded task flows. It has well-defined entry and exit points point and you can pass parameters to the task flow. Because a bounded task flow is a complete, self-contained piece of functionality, the same task flow can be used in several places in the application.

Your task flow should use page fragments instead of whole pages because a task flow using fragments can be embedded in a region on a page. When you are navigating through the task flow, only the content of the region changes – the contents of the page outside the region are not redrawn. This makes your ADF application look and feel like a desktop application instead of a series of web pages.

You need to include a task flow view in your task flow for every screen your user requirements call for – but you might need additional views. It is common to create separate search pages for the user to select data before you display it, or there might be overview pages before you show all the details.

Do not create pages for warnings, confirmation messages and other popup-style information. In older web applications, you might be presented with a page telling you that the record you requested deleted has indeed been deleted. With ADF, you can easily create real pop-up dialogs; you do not need to create separate pages in order to present this type of information to the user.

Draw **Control Flow Cases** for all valid transitions from one page to another. If some transitions need to be possible from every page or from many pages, remember that there is a **Wildcard Control Flow Rule** you can use to avoid cluttering up your task flow diagram with unnecessary arrows.

Finally, add any other elements you need – like method calls (to execute code between pages), routers (to conditionally go to one or another page), calls to other task flows, and so on.

Reviewing the task flows

Once you have built all the task flows, have someone else review them. The reviewer should preferably be an expert user—if this is not possible, get another team member to perform this review.

Agile ADF programming

Modern "agile" programming dispenses with some of the documentation that traditional software development uses. Instead, the programmers on the agile team work closely together with real users to ensure that they deliver what the users want. Even if you are not using an "agile" software development method, you can achieve some of its benefit by asking your users for feedback often.

The task flow diagrams are very visual and you might believe it should be immediately understandable to an end user. Still, do not just email it to the user. Sit down with a user (or use a screen sharing and voice call tool like GoToMeeting) to present your page flow to the user. This allows you to explain any objects in the flow that are not obvious to a non-programmer (method calls, routers and the like), as well as gather important feedback.

Of course, some of your task flows will consist of only one or two page fragments – in these cases, no review is necessary.

Creating the page fragments

With the page flow built and reviewed, you can start implementing the page fragments that make up the flow.

Before you start programming, talk to the other team members and stroll around the project room and have a look at all the other pages and page fragments used in the application (you did decorate your project room, didn't you?). This allows you to identify what similar elements exist in the application, and where you might be able to collaborate with another team member on a re-usable page fragment .

Implementing UI logic

By now, you have already implemented the application business logic in entity objects, view objects, application modules or the database. However, you might still have a bit of programming to do if the ADF user interface components do not meet your requirements out of the box. This part of the application is the user interface logic and is implemented in separate Java classes that are then registered in the task flow as **Managed Beans**. How to program user interface logic is outside the scope of this book.

Defining the UI test

The final development task is to define how to test what you have built. Some people advocate test-driven development, where the test is built before the code – however, this approach is not useful for user interface code. Firstly, the user interface is likely to go through a number of iterations before settling in the final form and secondly, because automated UI test tools work best if you record a session from a running application.

If your project is using an automated tool like Selenium, a tester should record the test cases as the final step in the development process. If you test the user interface manually, the tester needs to write a test script and check this document into your Subversion repository together with the code.

Reviewing the UI test

Irrespective of whether your test is automated or manual, someone else *must* review the test documentation.

For automated tests, this is simply a matter of having a tester run your script and possibly suggest additional test cases (did you remember to test all the error conditions? Your professional tester colleague will).

For manual tests, someone *who does not know the task flow you are implementing* must run your test. The developer who has been working on a task flow for days is *very* likely to accidentally skip some setup or intermediate steps because he is so familiar with the requirements and the solution.

Implementing Task Overview and Edit (UC008)

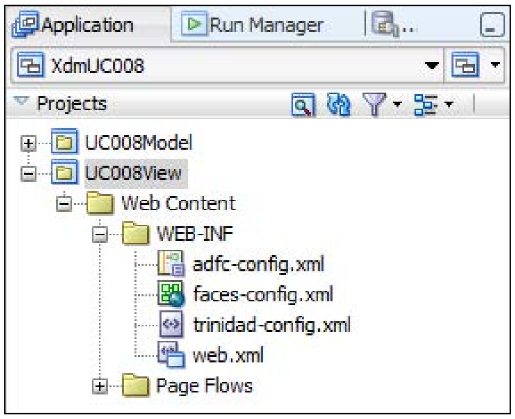
As an example of a subsystem in an enterprise application, we will build Task Overview and Edit (UC008). Later in this chapter, we will build Person Task Timeline (UC104) as another subsystem, and finally, we will integrate them together in a master application. In a real-life application, your subsystems will of course be larger than these simple one-screen use cases.

In this section, we will go fairly quickly over building of the business components and screens – you can refer back to *Chapter 1*, *The ADF Proof of Concept*, for more detailed descriptions.

Setting up a new workspace

Create a new **Fusion Web Application (ADF)** named after your subsystem, using a package name under your project base package, followed by the subsystem abbreviation.

For UC008 in the XDM application, use **XdmUC008** and the package name **com.dmcsol.xdm.uc008**. Remember to rename the model and view projects to **UC008Model** and **UC008View** and accept the default in the rest of the wizard:



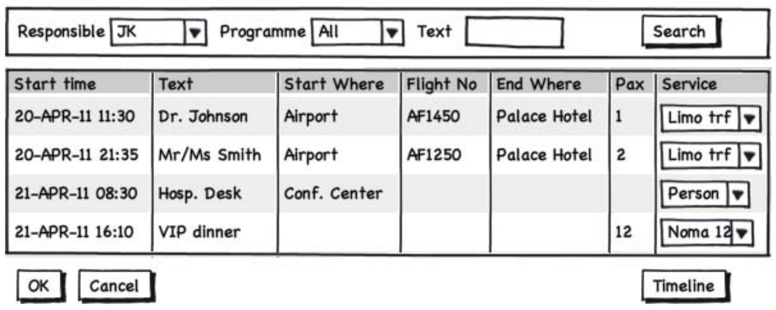
Getting the libraries

The model project needs the Common Code workspace for the framework extension classes and the Common Model for the entity objects. The view projects also needs Common Code for the framework extension classes, Common Model for the value list view objects and Common UI for the task flow template.

To include these ADF Libraries to the projects in your subsystem workspace, first select the **UC008Model** project and open the **XdmLib** node in the **Resource Palette**. Right-click on **adflibXdmCommonCode** and choose **Add to Project**. In similar way, add **adflibXdmCommonModel** to the **UC008Model** project, and add **XdmCommonCode**, **XdmCommonModel**, and **XdmCommonUI** to the **UC008View** project.

Creating business components

You will be getting the entity objects from the common model project—now you just need to figure out which view objects you need. To determine this, look at the screen design:



Start time	Text	Start Where	Flight No	End Where	Pax	Service
20-APR-11 11:30	Dr. Johnson	Airport	AF1450	Palace Hotel	1	Limo trf
20-APR-11 21:35	Mr/Ms Smith	Airport	AF1250	Palace Hotel	2	Limo trf
21-APR-11 08:30	Hosp. Desk	Conf. Center				Person
21-APR-11 16:10	VIP dinner				12	Noma 12

In this case, the screen design is just a mockup – a sketch showing the approximate layout and contents of the screen (and not all fields). In addition to the sketch, the following additional information is given:

The full list of fields to show in the table include:

* Start time and date (mandatory)

* Text (mandatory)

* Start location (mandatory)

* Flight no. (optional)

* End date and time (optional)

* End location (optional)

* No. of persons (optional)

* Service (mandatory)

* Comments (optional)

The list of services comes from the total catalog of elements.

The list of persons responsible (filter criteria) shows only those persons marked as program managers.

The list of programmes (filter criteria) shows all programmes.

The screen has two boxes, which often indicates two view objects. However, closer examination reveals that the topmost box doesn't really contain any data – it's just search criteria. So you need one view object showing tasks, including all required columns.

Your screen shows three value lists, but the team in charge of the common model should already have built ...`LOV` view objects for these. If you find that you need value list view objects that you cannot find in the common model project, ask the common model team to add them, so that they can be used across the entire application.

Starting work

If you are using a task management system integrated into JDeveloper (for example, Jira), the first thing to do when starting work on a new part of the application is to set the active task. Go to the **Team Navigator** as described in *Chapter 4*, *Productive Teamwork*, and run a query against your task repository. Find the task that specifies business components for the Task Overview and Edit use case, right-click it and choose **Make Active**.

Depending on the way you use task management, you might also open the task (listed at the top under the **Work Items** header in the **Team Navigator**), and change status to **Start Progress** or similar.

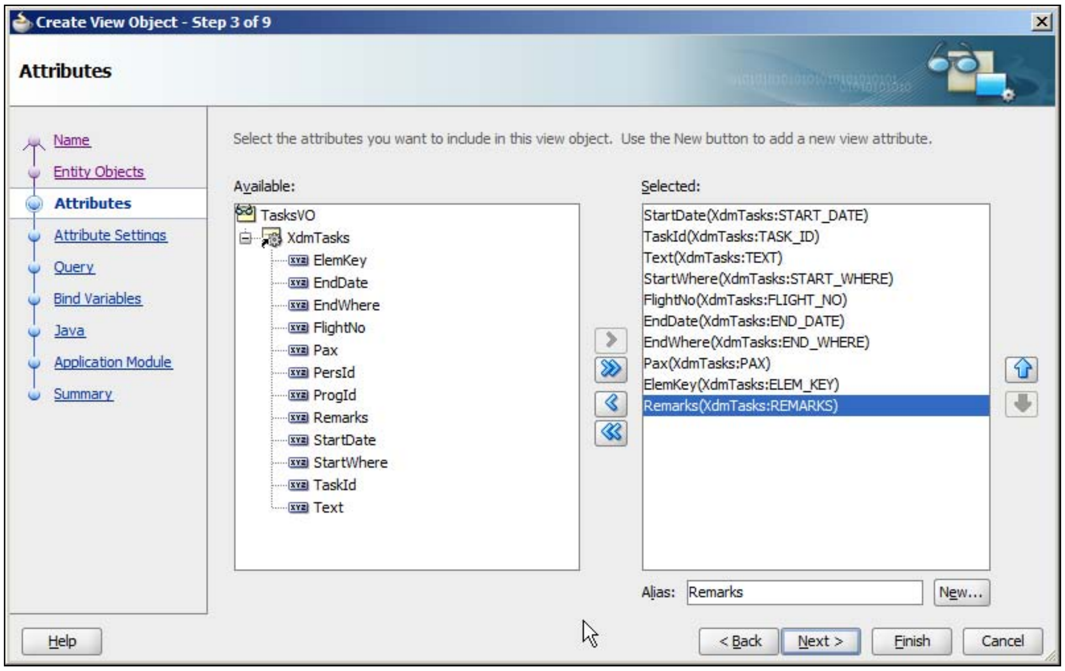
Building the main view object

To build the main view object, choose the **UC008Model** project in your subsystem workspace and choose **File | New** and then **Business Components**, **View Object**. Give your view object a name (for example, **TaskVO**), leave **Updatable access through entity objects** selected and click **Next**. In Step 2 of the wizard, all of the entity objects from the Common Model workspace should be available in the left-hand box.

No entity objects?

If you do not see any entity objects available, you might have forgotten to include Common Model ADF library in the model project of your subsystem workspace.

Select the **Task** entity object and in Step 3, select all the necessary attributes:



Select the attributes you want to include in this view object. Use the New button to add a new view attribute.

Available:

TasksVO
XdmTasks
ElemKey
EndDate
EndWhere
FlightNo
Pax
PersId
ProgId
Remarks
StartDate
StartWhere
TaskId
Text

Selected:

StartDate(XdmTasks:START_DATE)
TaskId(XdmTasks:TASK_ID)
Text(XdmTasks:TEXT)
StartWhere(XdmTasks:START_WHERE)
FlightNo(XdmTasks:FLIGHT_NO)
EndDate(XdmTasks:END_DATE)
EndWhere(XdmTasks:END_WHERE)
Pax(XdmTasks:PAX)
ElemKey(XdmTasks:ELEM_KEY)
Remarks(XdmTasks:REMARKS)

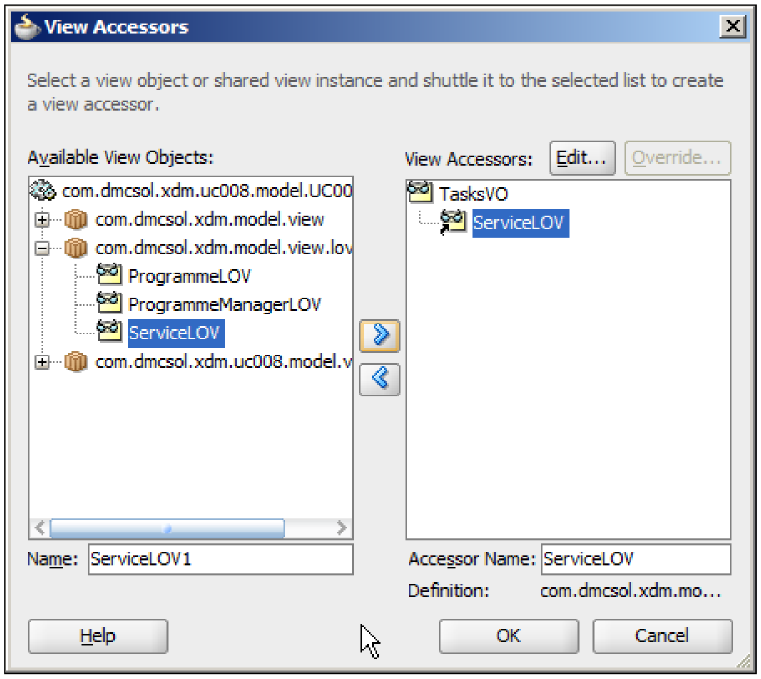
Alias: Remarks

In Step 5, select an **Order By** condition (`start_date`), and in Step 6 define the bind variables like you did in *Chapter 1*, *The ADF Proof of Concept*, (`pResponsible`, `pProgramme`, and `pText`). Go back to Step 5 of the wizard and add the following **Where** clause:

```
(:pResponsible is null or PERS_ID = :pResponsible)
and (:pProgramme is null or PROG_ID = :pProgramme)
and (:pText is null or upper(TEXT) like '%' || upper(:pText) || '%')
```

Then click **Finish** to create the `TaskVO` view object.

As in *Chapter 1*, *The ADF Proof of Concept*, you must also define a list of values for the service (ElemKey). Open the newly created **TaskVO** view object, go to the **Attributes** sub-tab and choose the `ElemKey` attribute. Create a new list of values with data source **ServiceLOV** (from the **com.dmcsol.xdm.view.lov** package):


View Accessors
Select a view object or shared view instance and shuttle it to the selected list to create a view accessor.
Available View Objects:
com.dmcsol.xdm.uc008.model.UC00
com.dmcsol.xdm.model.view
com.dmcsol.xdm.model.view.lov
ProgrammeLOV
ProgrammeManagerLOV
ServiceLOV
com.dmcsol.xdm.uc008.model.v
View Accessors:
Edit...
Override...
TasksVO
ServiceLOV
Name:
ServiceLOV1
Accessor Name:
ServiceLOV
Definition:
com.dmcsol.xdm.mo...
Help
OK
Cancel

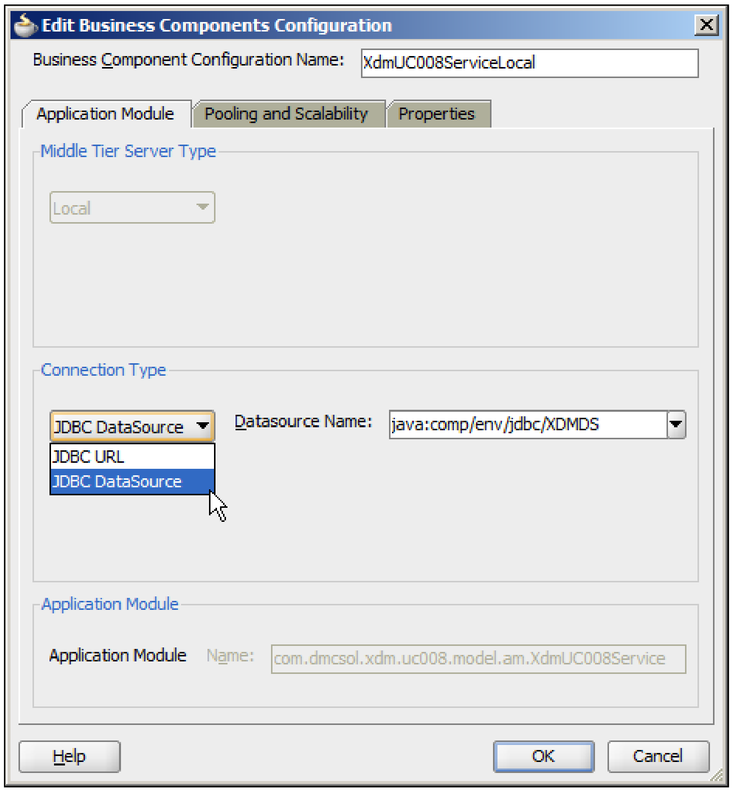
Use `ElemKey` as **List Attribute** and on the **UI Hints** tab choose to display the `Description` attribute and uncheck the **Include No Selection** checkbox.

Like in *Chapter 1*, *The ADF Proof of Concept*, set the format for the `StartDate` and `EndDate` attributes to **Simple Date** and format mask `dd-MMM-yy HH:mm`.

Building the application module

In order to make the `TaskVO` view object available to the UI developers, it must be part of an application module. Create an application module with the same name as the subsystem workspace, for example, `XdmUC008Service`, and include the view object in the data model.

When you have created the application module, right-click on it and choose **Configurations**. In the **Manage Configurations** dialog, select the `UC008ServiceLocal` configuration and click **Edit**. The **Edit Business Components Configuration** dialog opens:


Edit Business Components Configuration

Business Component Configuration Name: XdmUC008ServiceLocal

Middle Tier Server Type: Local

Connection Type: JDBC DataSource

Datasource Name: java:comp/env/jdbc/XDMDS

Application Module

Application Module Name: com.dmcsol.xdm.uc008.model.am.XdmUC008Service

Check that **Connection Type** is set to **JDBC Datasource**. This means that your application will only deploy with the name of a datasource it requires – so it will run on any application server as long as the application server administrator has defined a datasource with the right name. This dialog also allows you to set many ADF tuning parameters to optimize your ADF application, but these are outside the scope of this book.

Testing your business components

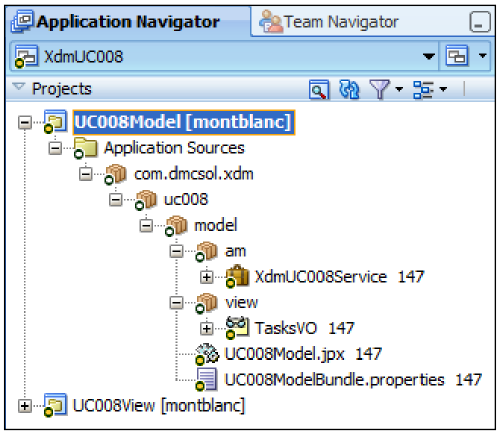
To test your view object, right-click on the application module and run the Business Components tester application to check that the correct data is shown, sorted as desired, and that you see value lists with the right values. In an enterprise project, you might decide to automate this testing with JUnit test cases. We will get back to testing in *Chapter 7, Testing Your Application*.

Checking in your code

When you have tested your business components, it is time to check them into Subversion. If you set up a connection to Subversion as described in *Chapter 4, Productive Teamwork*, you can now just choose **Versioning | Version Application** to start the Subversion import wizard.

In Step 2 of the wizard, remember to create a new folder in the trunk for your project with the same name as your workspace (for example, **XdmUC008**). Step through the wizard and remember to leave the **Perform Checkout** checkbox checked in order to immediately check out the application and continue working. You will see your import running in the SVN Console Log window.

All of the object in the **Application Navigator** should now have the little round Subversion "unmodified" marker on the icon:



Finishing the tasks

Once the code is successfully checked in to your source code repository, click on the **Active Work Item** (at the top of the **Work Items** list in the **Team Navigator**) to open the issue in JDeveloper. Provide a comment and mark the issue closed.

Creating the task flow

With the business components complete, we can consider the task flow – which pages will the user see. In the proof of concept in *Chapter 1, The ADF Proof of Concept*, we illustrated the concept of task flows by creating a task flow containing both UC008 and UC104. As discussed in *Chapter 2, Estimating the Effort*, a real enterprise application uses many separate bounded task flows – a simple use cases might have only one, while a complex use case can have a dozen or more.

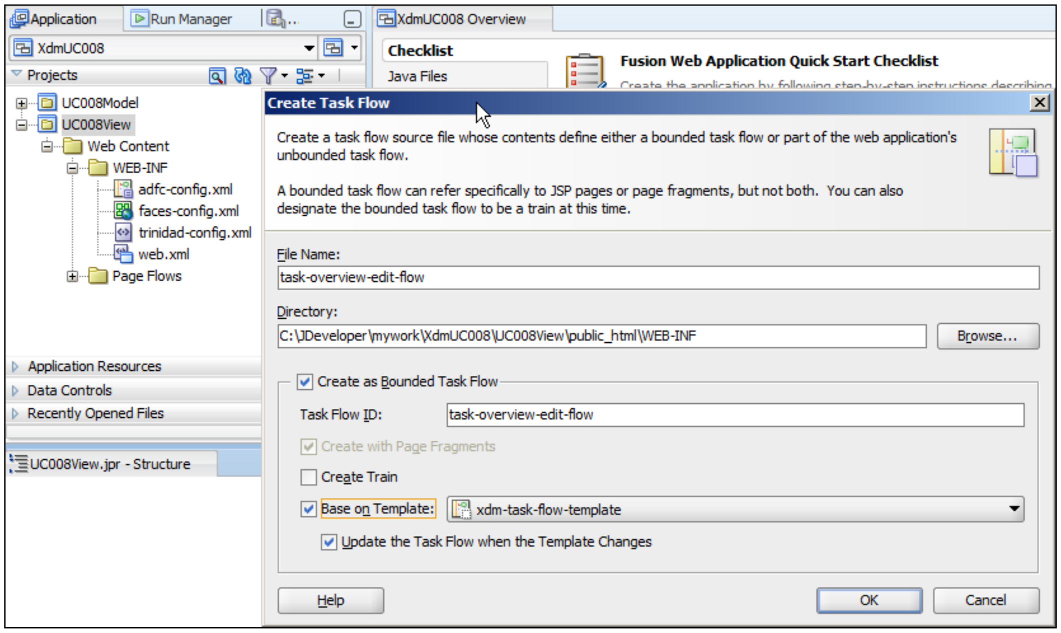
Our screen layout for this use case does not call for more than one screen – but we will still be building the bounded task flow to serve as a container for this use case. This gives us a uniform structure and the flexibility to add more screens later. In a web application, a use case will often include separate search screens, but in this case, the search capability is included directly on the page.

Starting work

Just like when you started working on the business components, first set the active task in the Team Navigator, and change status if your process calls for it.

Building the task flow

To build the task flow, select the **UC008View** project in the **XdmUC008** workspace and choose **File | New**, **Web Tier**, **JSF**, **ADF Task Flow**. Give your task flow a name, for example **task-edit-overview-flow**. Make sure that **Create with Page Fragments** is checked, and check **Base on Template**. If the list of templates does not contain the template you defined in the common UI workspace, you probably forgot to include the common UI library in the project:



Create Task Flow

Create a task flow source file whose contents define either a bounded task flow or part of the web application's unbounded task flow.

A bounded task flow can refer specifically to JSP pages or page fragments, but not both. You can also designate the bounded task flow to be a train at this time.

File Name: task-overview-edit-flow

Directory: C:\JDeveloper\mywork\XdmUC008\UC008View\public_html\WEB-INF

[✓] Create as Bounded Task Flow

Task Flow ID: task-overview-edit-flow

[✓] Create with Page Fragments

[ ] Create Train

[✓] Base on Template: xdm-task-flow-template

[✓] Update the Task Flow when the Template Changes

Task flows can either *copy* the content of the template, or *refer* to it. This is controlled by the **Update the Task Flow when the Template Changes** checkbox. It is recommended to leave this checkbox checked.

In the task flow editor, simply drop in a single **View** component from the **Component Palette** and call it **taskOverviewEdit**. Remember to check in your code and mark the task complete.

Creating the page fragments

To create the tasks page fragment, double-click on the **taskOverviewEdit** icon in the page flow to call up the **Create New JSF Page Fragment** dialog.

You will normally not base your page fragments on a template, so start from a blank page. The page fragments that make up the application will be used in regions on pages, and these pages will be based on page templates.

Layout

Your page will need three parts: The search panel at the top, the results table in the middle, and the OK/Cancel buttons at the bottom. Start by dragging in a **Panel Stretch Layout** from the **Component Palette** (in the **Layout** section).

A stretch target

Panel Stretch Layout and Panel Splitter are stretchable containers that also stretch the components they contain. These work well as outer containers, because they ensure that all space on the screen is used.

Data table

From the **Data Controls** panel in the left-hand side of the JDeveloper window, open the data control for your application module. Drag in the **Tasks** view object and drop it onto the center of the page (the **center** facet) as **Table | ADF Table**.

In the **Edit Table Columns** dialog, the `ServiceElemKey` attribute should be shown with an **ADF Select One Choice** component – this is because a list of values has been defined for this attribute in the view object. Allow the user to sort data in the table by checking the **Enable Sorting** checkbox.

The Panel Stretch Layout component is stretchable – that is, it will automatically stretch itself to fill all available space and will attempt to stretch the table as well. However, the table will not stretch unless you set the **ColumnStretching** property. Click in the **Text** column and make a note of its ID (in the **Property Inspector** – it will be something like **c7**). Then select the table (it is easiest to do this in the **Structure Panel** at the lower left) and set the **ColumnStretching** property to the ID of the **Text** column (for example, **column:c7**).

Search panel

The next part to build is the search panel above the data table. Since the fields and buttons we will be using for the search panel do not work well if you try to stretch them, we need to place them into a container that does *not* stretch its contents.

The non-stretch adapter

To convert between the stretchable layout containers that make up the outer part of your layout and the non-stretchable components that show your data, you use a Panel Group Layout. This component is special in that it will itself stretch (so it fits into a component that stretches its content), but it does not stretch its contents. You can consider the Panel Group Layout to be an "adapter" between stretchable and non-stretchable.

Drag a Panel Group Layout in from the **Component Palette** (**Layout** section) and drop it onto the top facet of the panel stretch layout (marked **top**). Set the **Layout** property to **Horizontal** to make the fields and buttons appear on the same line.

Then build the search panel like you did in *Chapter 1*, *The ADF Proof of Concept*:

- Expand the **Operations** node under the **Tasks** view object in the **Data Controls** panel. Open the operation **ExecuteWithParams** fully. Drag the **pResponsible** parameter onto the Panel Group Layout and drop it as a **Single Selection**, **ADF Select One Choice**. In the **Edit List Binding** dialog, leave **Base Data Source** at **variables** and add a new **List Data Source** based on **ProgrammeManagerLOV** in the **XdmLovServiceDataControl**. Choose **Initials** in the **Display Attribute** drop-down and set **"No Selection Item"** to **Blank Item (First of List)**. Then click **OK**. Use the **Property Inspector** to set the **Label** property to **Responsible**.
    - If you do not see the "…LOV" data sources, maybe you forgot to include the Common Model library in your view project?
- Next, drag the **pProgramme** parameter onto the Panel Group Layout next to the **pResponsible** drop-down as a **Single Selection**, **ADF Select One Choice**. Again leave the **Base Data Source** unchanged and add a new **List Data Source** based on the **ProgrammeLOV** data source. Use **ProgId** as **List Attribute** and set **Display Attribute** to **Name**. Set **"No Selection" Item** to **Blank Item (First of List)** and click **OK**. Finally, set the **Label** property to **Programme**.
- From the list of parameters, drop the **pName** parameter next to the **pProgramme** parameter and as **Text**, **ADF Input Text w/ Label**. Set the **Label** property to **Text**.

- Finally, drag the **ExecuteWithParams** operation (the node with the little green gearwheel icon) onto the page inside the Panel Group Layout, next to the three search criteria, as **Operation**, **ADF Button**. In the **Property Inspector**, change the **Text** property to **Search**.

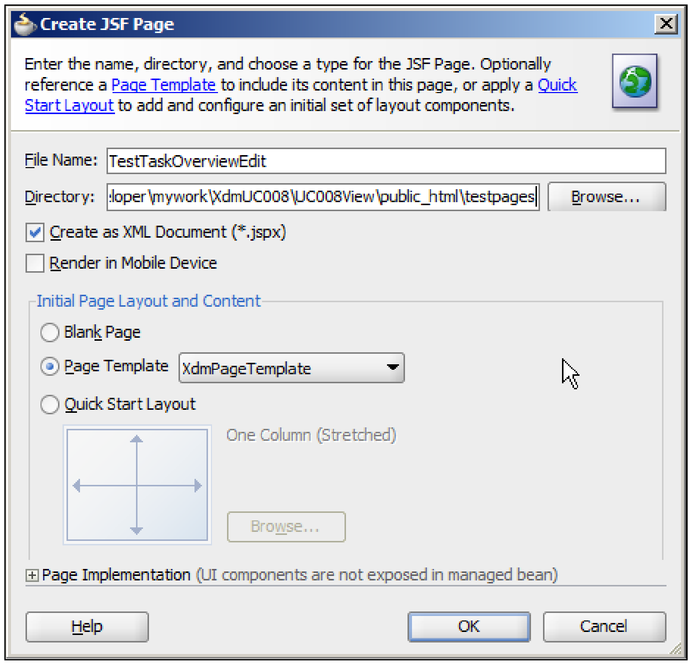
Translatable applications

For simplicity, we're defining user interface texts directly in the components here. If there is the slightest chance that your enterprise application will ever need to be translated into another language, you should use **Resource Bundles** to define your user interface texts. Refer to *Appendix A, Internationalization* A for more on creating translatable applications.

Running the page

Because your task flow is now based on page fragments, it cannot be run directly—you need to create a test page to run it.

Choose **File | New**, **Web Tier**, **JSF**, **JSF Page** and give the page a name like `TestTaskEditOverview`. In the Directory field, add `\testpages` at the end in order to place your test page in a directory separate from the rest of the application. You can base this test page on your page template to get a feel for how your final page will look:


Create JSF Page

Enter the name, directory, and choose a type for the JSF Page. Optionally reference a Page Template to include its content in this page, or apply a Quick Start Layout to add and configure an initial set of layout components.

File Name: TestTaskOverviewEdit

Directory: :loper\mywork\XdmUC008\UC008View\public_html\testpages

[✓] Create as XML Document (*.jspx)

[ ] Render in Mobile Device

Initial Page Layout and Content

( ) Blank Page

(●) Page Template: XdmPageTemplate

( ) Quick Start Layout

Browse...
One Column (Stretched)
Browse...
Page Implementation (UI components are not exposed in managed bean)
Help
OK
Cancel

When the page opens in the editor, open the **Page Flows** node to see your **task-edit-overview-flow** page flow. Drag it onto the content facet on the page and drop it as a **Region**. You will see a ghost image of your page – you cannot interact with it on this page. But you can right-click on the page and choose **Run** to see your task flow and page fragment in action.

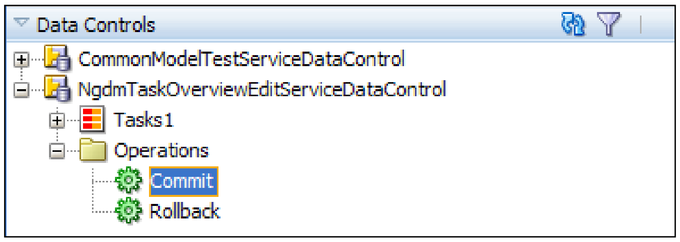
Regions and dynamic regions

There are two ways to include a page flow using page fragments in a page: As a **static region** (just called **Region** on the context menu) or as a **dynamic region**. If you use a static region, your page will always include exactly the bounded task flow you dropped onto the page. If you use a dynamic region, the ADF framework decides at runtime which task flow should be shown in the region. This allows you to swap out parts of the screen without the rest of the page refreshing, so your application feels more like a desktop application than a web site.

Now you should be able to filter data based on the drop-down lists and what you enter in the search field. Every time you click on the search button, the table should update to show only the records that satisfy the criteria.

OK and Cancel

The final elements you need to put on the page are the **OK** and **Cancel** buttons at the bottom, below the table. (We won't be adding the "Timeline" button that was used as an illustration of page flow navigation in *Chapter 1*, *The ADF Proof of Concept*.) These buttons will execute the **Commit** and **Rollback** actions that ADF provides at the application module level:


Data Controls
CommonModelTestServiceDataControl
NgdmTaskOverviewEditServiceDataControl
Tasks1
Operations
Commit
Rollback

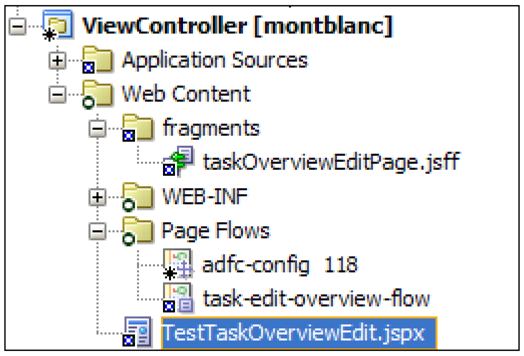
Before you drag these operations onto the page fragment, drop in a **Panel Group Layout** onto the bottom facet of the **Panel Stretch Layout**. Set the **Layout** property to **horizontal**.

Then drag the **Commit** and **Rollback** operations from the data control palette onto the **Panel Group Layout** as ADF Buttons. For both buttons, change the **Text** property (to **OK** and **Cancel**), and clear the **Disabled** property to ensure that both buttons are always active.

You can now run the page again and check that your buttons are placed correctly. Make some changes to the data, and click **OK**. Use the **Database Navigator** panel or a database tool to verify that your changes are committed to the database.

Checking in your code

When you look at the Application Navigator, you will see the nodes are marked with different subversion markers, not just the round "Unmodified":

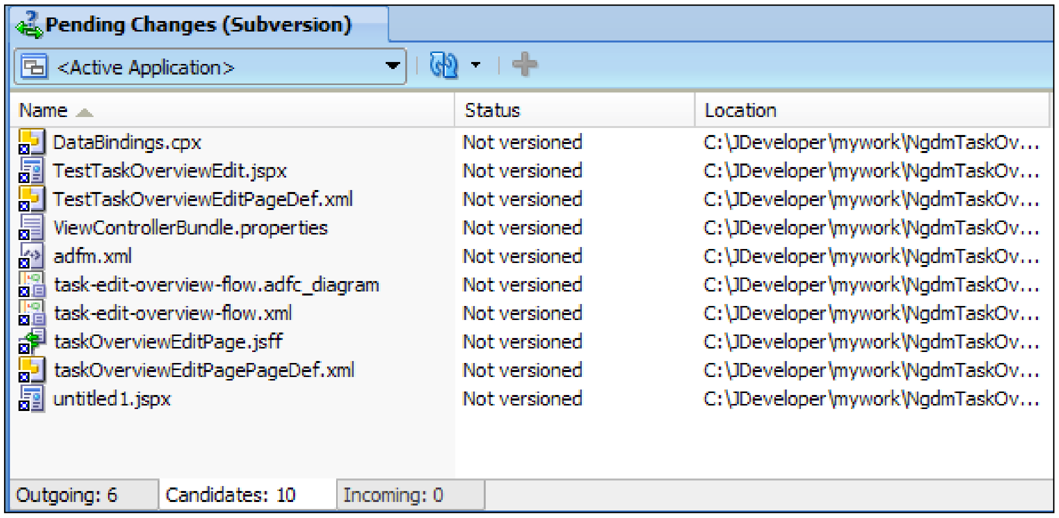


Some files are modified (marked with an *) and others are new and as yet unversioned (marked with a T).

Versioning new files

Remember that unless you have checked the checkbox **Automatically Add New Files on Committing Working Copy** (under **Tools | Preferences, Versioning, General**), Subversion will not automatically version new files. If you do not check this checkbox, you need to explicitly add them to Subversion to place them under version control.

If the **Pending Changes** panel is not shown (by default it appears below the main editing window), choose **Versioning | Pending Changes**:



Name	Status	Location
DataBindings.cpx	Not versioned	C:\JDeveloper\mywork\NgdmTaskOv...
TestTaskOverviewEdit.jspx	Not versioned	C:\JDeveloper\mywork\NgdmTaskOv...
TestTaskOverviewEditPageDef.xml	Not versioned	C:\JDeveloper\mywork\NgdmTaskOv...
ViewControllerBundle.properties	Not versioned	C:\JDeveloper\mywork\NgdmTaskOv...
adfm.xml	Not versioned	C:\JDeveloper\mywork\NgdmTaskOv...
task-edit-overview-flow.adfc_diagram	Not versioned	C:\JDeveloper\mywork\NgdmTaskOv...
task-edit-overview-flow.xml	Not versioned	C:\JDeveloper\mywork\NgdmTaskOv...
taskOverviewEditPage.jsff	Not versioned	C:\JDeveloper\mywork\NgdmTaskOv...
taskOverviewEditPagePageDef.xml	Not versioned	C:\JDeveloper\mywork\NgdmTaskOv...
untitled1.jspx	Not versioned	C:\JDeveloper\mywork\NgdmTaskOv...

Outgoing: 6 | Candidates: 10 | Incoming: 0

If you are not automatically adding new files, you will see a number of files listed as Candidates. These have not been placed under version control yet. Select them all and click on the green + to add them to Subversion. The **Candidates** tab empties, and the files are added to the **Outgoing** tab – you'll see the number of changes increase. Then choose **Versioning | Commit Working Copy** to commit your new and changed files to Subversion.

When you have checked in the UI code, you can mark the task complete (you did select an Active Work Item, didn't you?).

And remember that enterprise development also includes creating a test script:

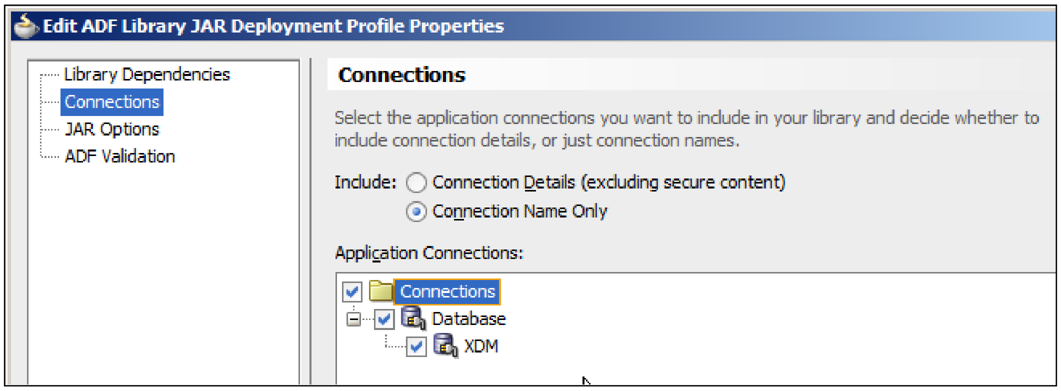
- A manual test script for a human to execute, or
- An automated test script using a tool like Selenium

We will discuss testing in more detail in the next chapter.

Deploying your UC008 subsystem

The final task is to deploy your subsystem as an ADF Library like you have done several times earlier. The library for a subsystem is created from the View project – JDeveloper automatically registers that the View project depends on the Model project, so all necessary objects from the Model project are included automatically.

Select the **UC008View** project and create a new **Deployment Profile** of type **ADF Library JAR File**. Give the deployment profile a name that includes your project abbreviation and the subsystem name, for example, **adflibXdmUC008**. In the **Edit ADF Library JAR Deployment Profile Properties**, choose to deploy **Connection Name Only**:


Edit ADF Library JAR Deployment Profile Properties

Connections

Select the application connections you want to include in your library and decide whether to include connection details, or just connection names.

Include: ( ) Connection Details (excluding secure content) [✓ Connection Name Only]

Application Connections:

This setting ensures that the individual subsystems don't include connection details but will use the connection defined in the master application.

Then right-click on the **UC008View** project and deploy to this profile.

Implementing person task timeline (UC104)

As an example of another subsystem, we will implement the second use case you saw in *Chapter 1*, *The ADF Proof of Concept* – the timeline showing the allocation of persons to tasks. In a real-life application, this would go into the same subsystem as UC008, but in order to demonstrate how to combine several subsystems into one master application at the end of this chapter, we will build a separate subsystem for UC104.

Setting up a new workspace

Again, we set up a separate subsystem workspace using **File | New** and then **Applications**, **Fusion Web Application (ADF)**. Each subsystem is implemented in a separate workspace, allowing you to divide the application between many team members without the implementation of one subsystem getting in the way of another.

Give your workspace a name that starts with the abbreviation for your enterprise application, following by the subsystem name. For the UC104 in the XDM application, use **XdmUC104** and the package name **com.dmcsol.xdm.uc104**. Name the model project **UC104Model** and the view/controller project **UC104View**.

Getting the libraries

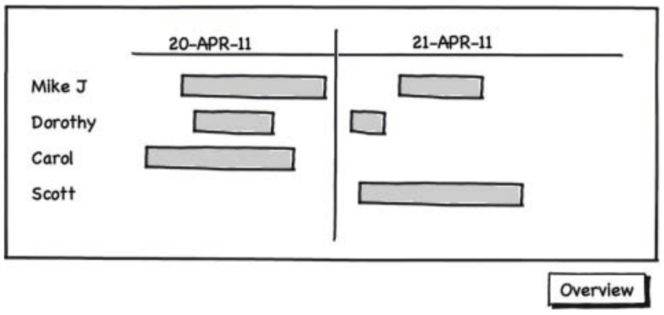
Select the **UC104Model** project and add the ADF Libraries **adflibXdmCommonCode** and **adflibXdmCommonModel**, and select the **UC104View** project and add **adflibXdmCommonCode**, **adflibXdmCommonModel**, and **adflibXdmCommonUI**.

Creating business components

Before you start working, find the task you need to work on from the Team Navigator and make it active.

Creating view objects for scheduling

To determine the view objects you need, you look at the screen for the person task timeline:



It is not immediately obvious which view objects we need for a component like this. One useful way of looking at the data is to think of **iterators** – what are the data sets that we must be looping through in order to build the screen.

For the screen above, it is clear that we must be looping over persons as we go down the left-hand column. That means we will need a `Person` view object. Additionally, we must be looping over tasks as we go across to build up the sequence of tasks for each person: We therefore need a `Task` view object as well.

Building the persons view object

We can see that we will need a view object showing persons – just the first and last name is necessary, and since we will not be changing data from this screen, the view object can be read-only.

Tell the framework what you know

When you know that you will not have to update data through a view object, tell the ADF framework by deselecting the **Updatable** checkbox. The more information you give ADF, the better it can manage performance for you.

In the **UC104Model** project, choose **File | New** and then **Business Tier**, **ADF Business Components**, **View Object**. Give a name like **PersonVO** and leave data source at **Updatable access through entity objects**.

Read-only or updatable?

The options for data source type are not well named in the version of JDeveloper that was current at the time of writing: One says "updatable" and the other says "read-only". In fact, you should always use **Updatable access through entity objects** even if you do not plan on updating anything. You can deselect the **Updatable** checkbox later in the wizard.

Add the **Persons** entity object and *deselect* the **Updatable** checkbox. Add the **FirstName** and **LastName** attributes (the **PersonId** is automatically included) and order by **last_name**, **first_name**.

Building the tasks view object

Create another view object, giving it the name **TaskVO**. Base it on the **Tasks** entity object, and again deselect **Updatable**. You just need the **StartDate** and **EndDate** attributes (the **TaskId** is automatically included). Add a where clause so that the view object will only show tasks with both a start and an end date, using the following:

```
start_date is not null and end_date is not null
```

Building the master-detail link

Because tasks and persons are related, you also need to define the relationship in the form of a view link. Select **File | New** and then **Business Tier**, **ADF Business Components**, **View Link**. Give your view link the name **PersonTaskLink**.

In step two of the wizard, expand the **PersonVO** node on the left and choose **PersTaskFkAssoc** as the left-hand side of the link. On the right, expand the **TaskVO** node and again choose **PersTaskFkAssoc**, this time as the right-hand side of the link. Then click **Add**, **Next**, and **Finish**.

Building the MinMaxDate view object

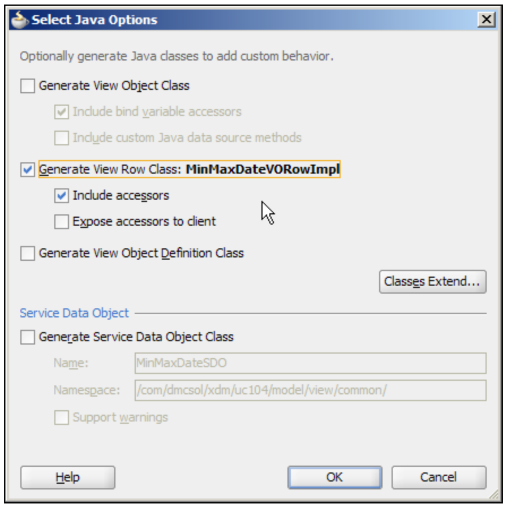
As you might remember from *Chapter 1*, *The ADF Proof of Concept*, the Gantt chart component does not automatically scale to the time data it presents, so you have to set **startTime** and **endTime** attributes. To retrieve these values, we will create an SQL Query view object.

Create a view object, giving it the name **MinMaxDateVO** and select Read-only access through SQL Query. In step 2, enter the following query:

```
Select min(start_date) - 1 as min_start_date,
      max(end_date) + 1 as max_end_date
from   xdm_tasks
```

Click the **Test** button to ensure that the query is valid and then **Next** several times and **Finish**. This creates a view object with one row, containing a `MinStartDate` attribute one day before the earliest start date in the table, and a `MaxEndDate` attribute one day after the latest end date.

Because the Gantt chart component requires the start and end dates to be instances of `java.util.Date` (and not the default `oracle.jbo.domain.Date`), we need to create a Java class for this view object to perform this conversion. Choose the **Java** sub-tab and click the pencil icon to generate a Java class for this view object. Check the checkbox **Generate View Row Class** and leave **Include accessors** checked:



Select Java Options

Optionally generate Java classes to add custom behavior.

- [ ] Generate View Object Class
  - [✓] Include bind variable accessors
  - [ ] Include custom Java data source methods
- [✓] Generate View Row Class: **MinMaxDateVORowImpl**
  - [✓] Include accessors
  - [ ] Expose accessors to client
- [ ] Generate View Object Definition Class

Service Data Object

- [ ] Generate Service Data Object Class
  - Name: MinMaxDateSDO
  - Namespace: /com/dmcsol/xdm/uc104/model/view/common/
  - [ ] Support warnings

When you click OK, the class `MinMaxDateVORowImpl.java` is created. Open this class and change the `getMinStartDate()` and `getMaxEndDate()` methods to look like this:

```
…
/**
 * Gets the attribute value for the calculated attribute
 * MinStartDate.
 * @return the MinStartDate
 */
public java.util.Date getMinStartDate() {
  return ((Date)getAttributeInternal(MINSTARTDATE)).getValue();
}
…
/**
 * Gets the attribute value for the calculated attribute
 * MaxEndDate.
 * @return the MaxEndDate
 */
public java.util.Date getMaxEndDate() {
  return ((Date)getAttributeInternal(MAXENDDATE)).getValue();
}
…
```

In the code above, the signature of these two methods has been changed to return a `java.util.Date` instead of the default `oracle.jbo.domain.Date`, and a `getValue()` call has been added to convert the `oracle.jbo.domain.Date` attribute value to a `java.util.Date`.

Building the application module

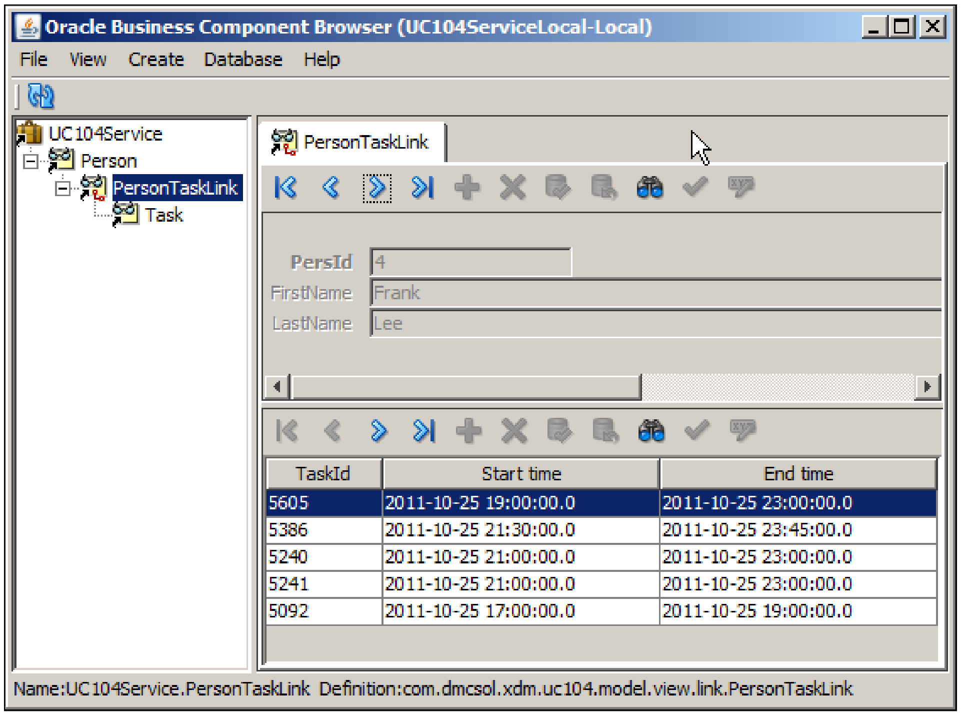
Create an application module called **XdmUC104Service** to collect the data model for the timeline screen.

In Step 2 of the wizard, first select the **PersonVO** view object on the left. Correct the **New View Instance** field to **Person** and click the **>** button to add a view object instance to the application module. Then select the new **Person** view instance on the right and the node **TaskVO** `via` **PersonTaskLink** on the left. In the **New View Instance** field, change the value to **Task** and click the **>** button to create a new view instance *as a child of* **Person**. Remember that you need to point to **TaskVO** via a link to get a detail view – if you just choose the **TaskVO** without using the link, there would be no relationship between persons and tasks, and the Gantt chart component would not work. Click **Finish** to close the wizard. Finally, add the **MinMaxDateVO** view object.

Once you have created the application module, verify that **Connection Type** is set to **JDBC Datasource** (right-click **Configurations**, **Edit**).

Testing your business components

To test your application module, you can right-click on the **XdmUC104Service** application module node in the application navigator and choose Run from the context menu. Double-click on the view link to see master and detail records together and verify that data looks as expected:


Oracle Business Component Browser (UC104ServiceLocal-Local)
File View Create Database Help
UC104Service
Person
PersonTaskLink
Task
PersonTaskLink

PersId	4
FirstName	Frank
LastName	Lee

TaskId	Start time	End time
5605	2011-10-25 19:00:00.0	2011-10-25 23:00:00.0
5386	2011-10-25 21:30:00.0	2011-10-25 23:45:00.0
5240	2011-10-25 21:00:00.0	2011-10-25 23:00:00.0
5241	2011-10-25 21:00:00.0	2011-10-25 23:00:00.0
5092	2011-10-25 17:00:00.0	2011-10-25 19:00:00.0

Name:UC104Service.PersonTaskLink Definition:com.dmcsol.xdm.uc104.model.view.link.PersonTaskLink

Finishing the tasks

The final step after your own test is as always to check the code in to your source code repository, select the Active Work Item and mark the corresponding issue closed.

Building the Task Flow

The UC104 Person Task Timeline use case also uses only one screen, but you should still create a one-page bounded task flow for it.

Set the active work item, go to the **UC104View** project in the **XdmUC104** workspace and create a new ADF Task Flow. Give it a name, for example, **person-timeline-flow**. **Create with Page Fragments** should be checked and the task flow should be based on **xdm-task-flow-template**.

In the task flow editor, simple drop in a single **View** component from the **Component Palette** and call it **personTimeline**.

Then check in your code and mark the task complete.

Building the page

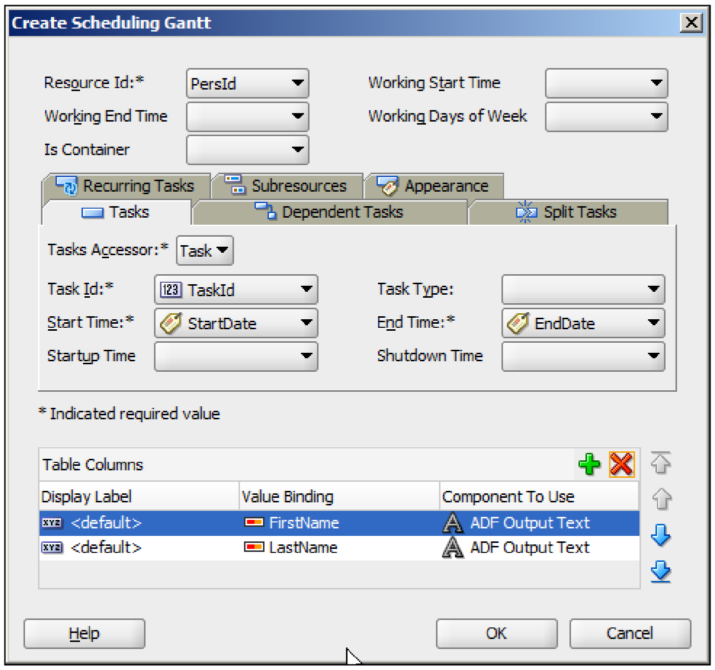
To create the scheduled tasks page fragment, double-click on the **personTimeline** in the task flow. Again, the page fragment does not have to be based on a template because it will be used on a page that has all the common elements (logos, menu bar, and so on)

As in most cases, this page starts with a stretchable outer component. Drag a **Panel Stretch Layout** onto the page from the **Component Palette**.

Adding a Gantt chart component

As you might remember from *Chapter 1*, *The ADF Proof of Concept*, the component that implements the graphic representation of tasks assigned to persons is a Gantt chart of type Scheduling.

To create the page, open the **UC104ServiceDataControl** node and drag the **Persons** view instance onto the **center** facet and drop it as **Gantt**, **Scheduling**. The **Create Scheduling Gantt** dialog appears:



Resource Id:*	PersId	Working Start Time	
Working End Time		Working Days of Week	
Is Container			

Tasks Accessor:* Task

Task Id:*	TaskId	Task Type:	
Start Time:*	StartDate	End Time:*	EndDate
Startup Time		Shutdown Time	

* Indicated required value

Table Columns

Display Label	Value Binding	Component To Use
<default>	FirstName	ADF Output Text
<default>	LastName	ADF Output Text

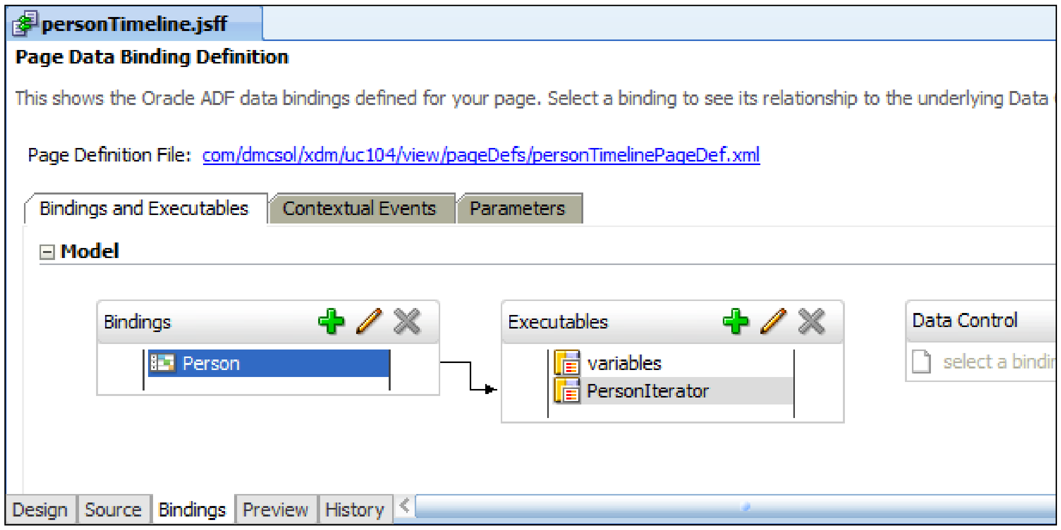
Set the fields in this dialog as follows:

- **Resource Id**: **PersId**
- **Tasks Accessor: Task**
- **Task Id: TaskId**
- **Start Time: StartDate**
- **End Time: EndDate**

Under **Table Columns**, remove the extra **PersId** column, leaving only **FirstName** and **LastName**. Then click **OK** to see a graphical representation of a scheduling Gantt chart.

Defining start and end time

The **MinStartDate** and **MaxEndDate** attributes can be found in the **Data Control Panel** under the **MinMaxDate** view instance, but you cannot simply drag them onto the attribute we need. We will have to create a binding manually. To do this, change to the **Bindings** tab:



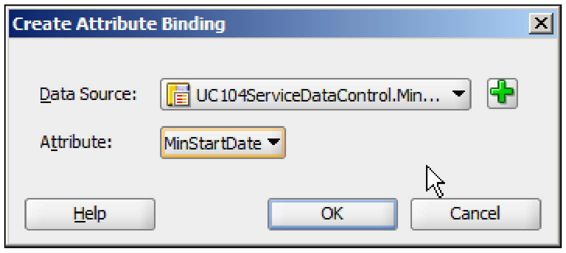
personTimeline.jsff

Page Data Binding Definition

This shows the Oracle ADF data bindings defined for your page. Select a binding to see its relationship to the underlying Data

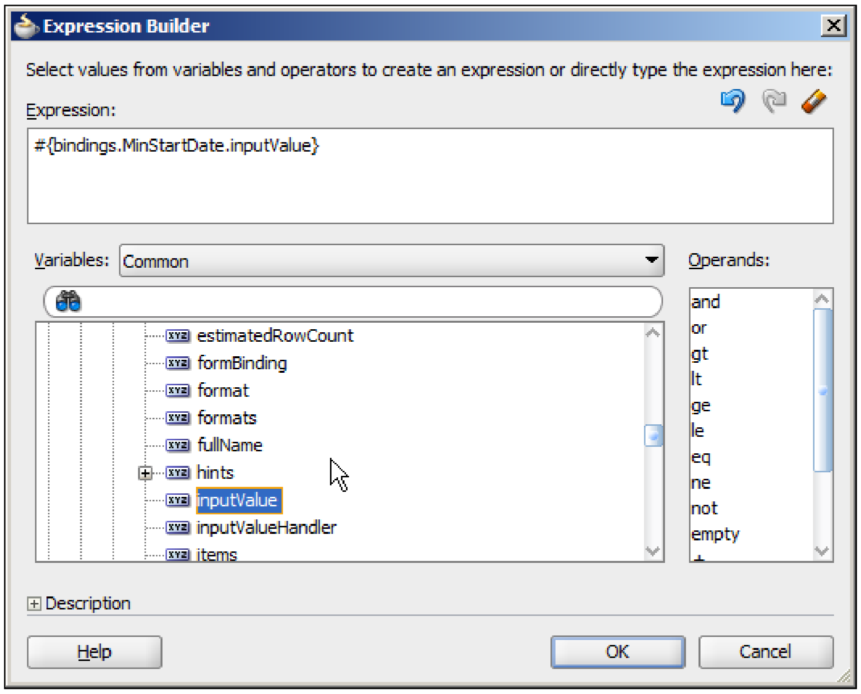
Page Definition File: com/dmcsol/xdm/uc104/view/pageDefs/personTimelinePageDef.xml

Click on the green plus sign next to **Bindings** and choose to create an **attributeValues** binding. In the **Create Attribute Binding** dialog, click the green plus sign to create a new **Data Source** and select the **MinMaxDate** view instance. Then select the **MinStartDate** attribute:



Add another **attributeValues binding** with **MinMaxDate** as **Data Source** and **MaxEndDate** as **Attribute**.

With the bindings for start and end date created, click on the **Design** tab to return to the person timeline page fragment and select the **dvt:schedulingGantt** component in the **Structure Panel**. Then click on the triangle next to the **StartDate** attribute and choose the expression builder. Open **ADF Bindings** and then **bindings**. Select the **MinStartDate** attribute and then **inputValue**:


Expression Builder
Select values from variables and operators to create an expression or directly type the expression here:
Expression:

```
#{bindings.MinStartDate.inputValue}
```

Set the **EndTime** property to **#{bindings.MaxEndDate.inputValue}** in similar way.

At runtime, the Gantt chart component will retrieve the start and end date from the binding, which connects the component to the **MinMaxDate** view object.

Running the page

Just like the Task Overview and Edit task flow, this task flow is based on page fragments and cannot be run directly. Create a new JSF Page with a title like **TestPersonTimeline** and place it in the `testpages` subdirectory of `public_html`.

When the page opens in the editor, drop the **person-timeline-flow** page flow onto the content facet on the page as a region. You can then right-click on the page and choose **Run** to see your task flow and page fragment in action.

Checking in your code

If the Pending Changes panel is not shown, choose **Versioning | Pending Changes** to call it up. Go to the **Candidates** tab and add all the new files to Subversion, and then choose **Versioning | Commit Working Copy** to commit your new and changed files to Subversion.

Now you can mark the task complete, and all that remains is to write a test script.

Deploying your UC104 subsystem

Like you did for the UC008 subsystem, you need to deploy the UC104 subsystem as an ADF Library, this time called **adflibXdmUC104**. Remember to choose **Connection Name Only** when defining the ADF Library deployment profile.

Building the master application

In this chapter, we have built two very small subsystems and deployed them as ADF libraries – now we will create the master application that brings the subsystems to-gether to the final application. A real-life enterprise application will have a number of subsystems, and these will be much larger, but the same principle applies.

In addition to the task flows from the subsystems, the master application will contain:

1. The master application page
2. A dynamic region for the task flows
3. A menu for selecting task flows
4. A bit of user interface code to tie everything together

Setting up the master workspace

Create a new workspace of type **Generic Application**, naming it **XdmMaster** and assigning an application package prefix of **com.dmcsol.xdm**. Also name the project **XdmMaster** and include the following technologies:

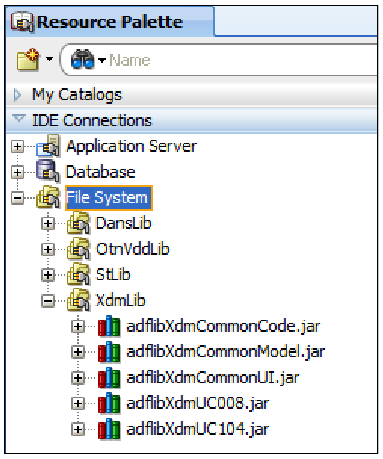
1. **ADF Business Components**
2. **ADF Faces**
3. **HTML**
4. **XML**
5. **Ant**

In the project properties dialog, choose the **Java EE Application** node and shorten the content of the **Java EE Web Application Name** and **Java EE Web Context Root** fields to just **Xdm**.

Getting the Libraries

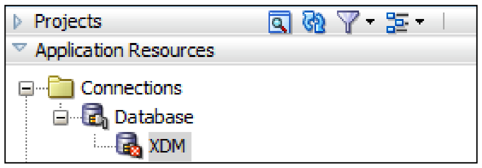
As described in the section on ADF Library workflow at the beginning of this chapter, the Build/Deployment manager collects the subsystem workspaces, has them tested and releases them for use. For the purposes of the example in this chapter, take the role of Build/Deployment manager and copy `C:\JDeveloper\mywork\XdmUC008\UC008View\deploy\adflibXdmUC008.jar` to `C:\JDeveloper\XdmLib`. Also copy `C:\JDeveloper\mywork\XdmUC104\UC104View\deploy\adflibXdmUC104.jar` to `C:\JDeveloper\XdmLib`.

You should now have five ADF Libraries in your in the Resource Palette under **XdmLib**:



Add them all to the **XdmMaster** project in the **XdmMaster** workspace.

Now open the **Application Resources** heading in the **Application Navigator**. You will see that it contains a database connection called **XDM**, but it is marked with a red x.

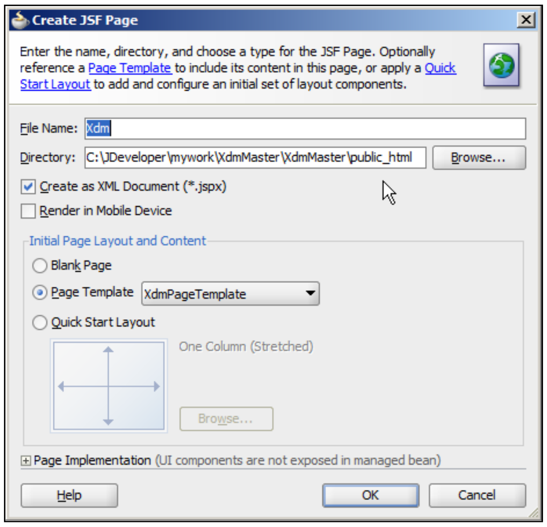


This is because the ADF Libraries you have added contain an XDM connection, but you only included the name when creating the ADF Library. Here, in the master application, you must define the connection properties. You do this by right-clicking the connection, choosing **Properties** and filling in the dialog box.

Create the master page

An ADF enterprise application will contain many bounded task flows, each containing many page fragments – but it will have few pages, possibly only one. You need one page for every direct access point your application needs – if you want three different entry points to the application with three different URLs, you need three pages.

In the example in this chapter, we will create only one page. To create the page, choose **File | New** and then **Web Tier**, **JSF Page**. Give it the name **Xdm** and choose to base it on the **XdmPageTemplate**:


Create JSF Page

Enter the name, directory, and choose a type for the JSF Page. Optionally reference a Page Template to include its content in this page, or apply a Quick Start Layout to add and configure an initial set of layout components.

File Name: Xdm

Directory: C:\JDeveloper\mywork\XdmMaster\XdmMaster\public_html

☑ Create as XML Document (*.jspx)

☐ Render in Mobile Device

Initial Page Layout and Content

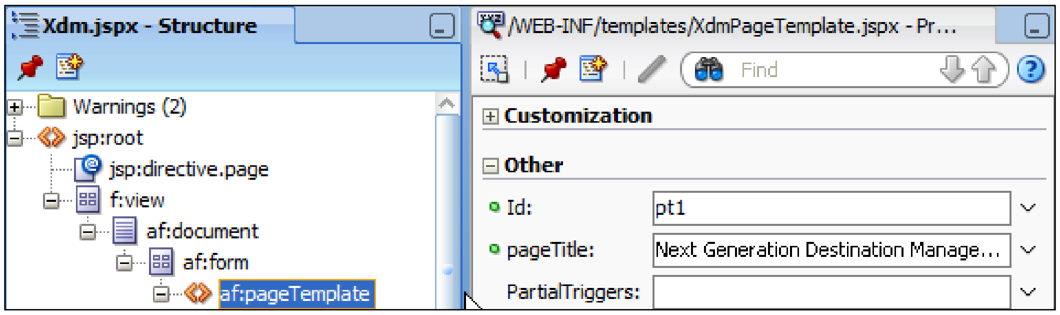
○ Blank Page

● Page Template: XdmPageTemplate

○ Quick Start Layout: One Column (Stretched)

Page Implementation (UI components are not exposed in managed bean)

In the **Structure Panel**, select the **af:pageTemplate** node. Among the properties in the **Property Palette**, you now see the **pageTitle** variable you defined in the page template. Set the value to **Next Generation Destination Management**.



Id: pt1
pageTitle: Next Generation Destination Manage...

Create the layout

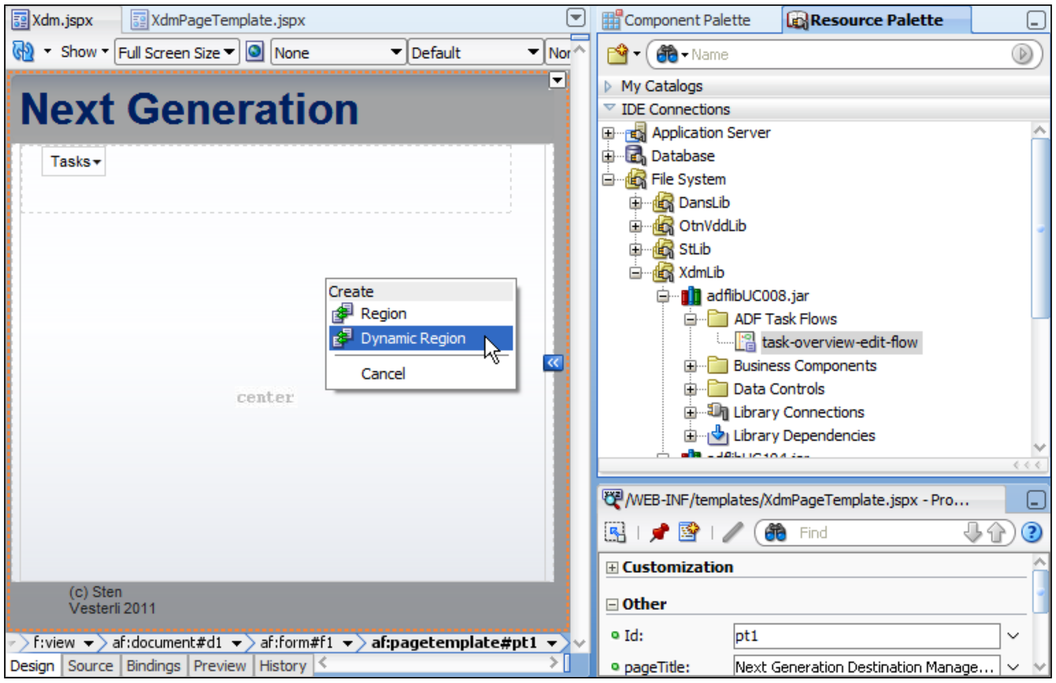
Drop a **Panel Stretch Layout** component on the **main** facet of the page. This component has five facets, but we only need the **top** and **center** facets. To remove the unnecessary facets from view, right-click on the component and choose **Facets – Panel Stretch Layout** and deselect the **start** facet. Repeat to deselect **end** and **bottom** facets.

Adding the menu

To build the menu, first drop a **Panel Menu Bar** component on the top facet. Then drop a **Menu** component on the menu bar, setting the **Text** property to **Tasks**. Finally, drop two **Menu Item** components onto the **Tasks** menu, setting the **Text property to Overview/Edit** and **Timeline**, respectively.

Creating a dynamic region

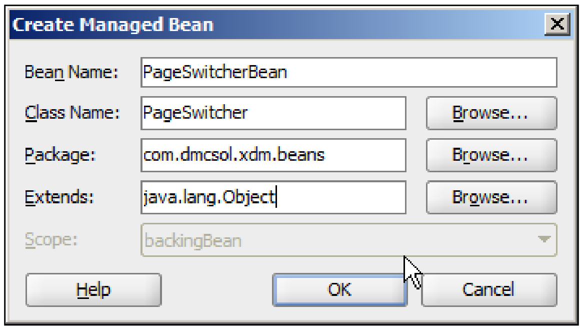
To create a dynamic region on the master application page, open the **Resource Palette** and then the **adflibXdmUC008** library. Expand the **ADF Task Flows** node and drag the **task-overview-edit-flow** onto the **center** facet of the **Panel Stretch Layout** on the page:



Id: pt1

pageTitle: Next Generation Destination Manage...

You will be prompted to select or create a managed bean that will provide the URL to the task flow to be displayed in the dynamic region. Click the green plus sign to create a new managed bean and fill in the **Create Managed Bean** dialog as shown next:


Create Managed Bean

Bean Name:	PageSwitcherBean
Class Name:	PageSwitcher
Package:	com.dmcsol.xdm.beans
Extends:	java.lang.Object
Scope:	backingBean

Click **OK** twice to create and select the bean. You can also just click **OK** in the **Edit Task Flow Binding** dialog – if your task flow had taken parameters, this dialog would be where you'd define values for these.

These steps do four things:

1. Create a new Java class with the package and name you specify.
2. Define the Java class as a managed bean in the unbounded task flow (in the `adfc-config.xml` file).
3. Create a task flow binding in the data bindings for the page (`XdmPageDef.xml`).
4. Set the **Value** attribute of the dynamic region to point to the task flow binding.

Managed beans

Managed beans are Java classes that are instantiated by the JSF framework as needed. JSF enforces separation between user interface components and user interface logic—the JSF page contains the components, and the managed beans contain the user interface logic.

Understanding the dynamic region

If you run the page now, you'll see the Task Overview and Edit task flow. Let us follow the execution flow to understand how these pieces fit together.

1. Whenever the ADF framework has to show the page containing the dynamic region, it will execute the task flow binding indicated by the **Value** attribute of the dynamic region. In the XDM example, this value is **#{bindings.dynamicRegion1.regionModel}**.
2. If you click the **Bindings** tab on the **Xdm.jspx** page, you will see the task flow binding called **dynamicRegion1**. In the **Property Palette**, you can read the **taskFlowId** attribute to see where the task flow for the binding comes from. In the XDM example, this is **${backingBeanScope.PageSwitcherBean.dynamicTaskFlowId}**.
3. This is a reference to a managed bean defined on the unbounded task flow. If you open the **adfc-config.xml** file, go to the **Overview** tab and then the **Managed Beans** sub-tab, you will find a bean with a scope of **BackingBean** and the name **PageSwitcherBean**. On this screen, you can also see the actual class implementing the managed bean. In the XDM example, this is **com.dmcsol.xdm.beans.PageSwitcher**.
4. If you open this class, you will see that it has a method **getDynamicTaskFlowId**, corresponding to the binding you found in step 2 above. Whatever this method returns is used to look up the task flow to be displayed in the dynamic region.

Additional code for task flow switching

To switch between different task flows inside the dynamic region, we need some way of making the `getDynamicTaskFlowId()` method return different values, and some way of setting these values. The solution we will implement consists of 4 parts:

1. Another managed bean to store the value across page requests.
2. A way to access this second bean from the first.
3. A way to set values in this second bean from the user interface.
4. A way to make the region redraw itself when needed.

Storing the selected task flow value

The existing managed bean has a scope of **backingBean**. This is a short scope, only valid for the duration of one request from the browser to the server – so it cannot *store* the selected task flow as the user is working with the application. Instead, we need a managed bean with a **session** scope. Values in a session-scoped managed bean endure until the user closes his browser, so such a bean *can* store the selected task flow.

To store the selected task flow, create a new Java class called `UiState` in package `com.dmcsol.xdm.beans`. At the most basic, this bean only needs to store a reference to the currently selected task flow; this can be done with the following code:

```
package com.dmcsol.xdm.beans;

public class UiState {
private String currentTF = "/WEB-INF/task-overview-edit-flow.
xml#task-overview-edit-flow";
public void setCurrentTF(String s) {
currentTF = s;
}
public String getCurrentTF() {
return currentTF;
}}
```

This class must be added to the unbounded task flow as a managed bean. To do this, open the **adfc-config.xml** file (under **Web Content / WEB-INF**), choose the **Overview** tab and then the **Managed Beans** sub-tab. Click the green plus sign at the top and add a new managed bean with the name **UiStateBean**, **class com.dmcsol.xdm.beans. UiState** and scope **session**.

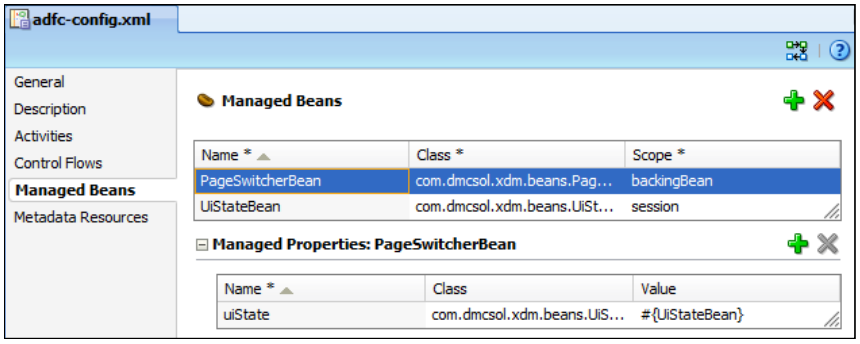
This chapter uses the simplest possible example to illustrate the use of dynamic regions, and the UiState class simply stores a Java String value. Obviously, this is not a robust solution, breaking as soon as a string is misspelled. In a real enterprise application, all task flows should be stored in a data structure, and a key should be used to look up the task flow.

Accessing the session bean from the backing bean

We can use the JSF functionality of **Managed Properties** to make the `UiState` session bean available to the `PageSwitcher` backing bean. To do this, change the `PageSwitcher` class to look like this:

```
package com.dmcsol.xdm.beans;

import oracle.adf.controller.TaskFlowId;

public class PageSwitcher {
private UiState currentUiState;
public TaskFlowId getDynamicTaskFlowId() {
return TaskFlowId.parse(
currentUiState.getCurrentTF());
}
public void setUiState(UiState state) {
currentUiState = state;
}}
```

This class now has a private variable of class `UiState`, and a corresponding `setUiState` method. We can ask JSF to automatically set `UiState` whenever the `PageSwitcherBean` is instantiated by defining a **Managed Property**. In the **adfcconfig.xml** file on the **Managed Beans** sub-tab, first select the **PageSwitcherBean** at the top and then click the lower green plus sign. The managed property name should be **uiState** (matching the `setUiState()` method), the class should be **com.dmcsol. xdm.beans.UiState** and the value should be **#{UiStateBean}**.


adfc-config.xml

General
Description
Activities
Control Flows
Managed Beans
Metadata Resources

Managed Beans

Name *	Class *	Scope *
PageSwitcherBean	com.dmcsol.xdm.beans.Pag...	backingBean
UiStateBean	com.dmcsol.xdm.beans.UiSt...	session

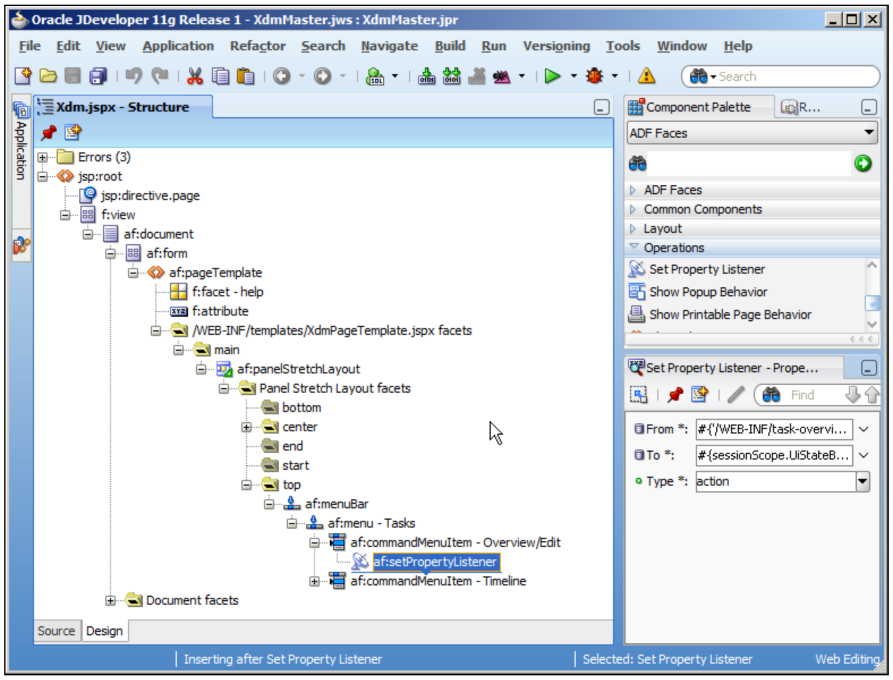
Managed Properties: PageSwitcherBean

Name *	Class	Value
uiState	com.dmcsol.xdm.beans.UiS...	#{UiStateBean}

Setting the task flow values

The final thing we need is to set the **UiState** value to select one or the other task flow. This can be done with a **Set Property Listener** operation that you can drag in from the **Component Palette**. In the XDM user interface, the switching of task flows is done with **af:commandMenuItem** elements, but other command elements like buttons and links can also be used.

To set the value, open the menu in the **Structure Panel** and expand the **Operations** node in the **Component Palette**. Then simply drag a **Set Property Listener** operation onto the **af:commandMenuItem**:


Oracle JDeveloper 11g Release 1 - XdmMaster.jws : XdmMaster.jpr

Xdm.jspx - Structure

```
Errors (3)
jsp:root
  jsp:directive.page
  f:view
    af:document
      af:form
        af:pageTemplate
          f:facet - help
          f:attribute
          /WEB-INF/templates/XdmPageTemplate.jspx facets
            main
              af:panelStretchLayout
                Panel Stretch Layout facets
                  bottom
                  center
                  end
                  start
                  top
                    af:menuBar
                      af:menu - Tasks
                        af:commandMenuItem - Overview/Edit
                          af:setPropertyListener
                        af:commandMenuItem - Timeline
      Document facets
```

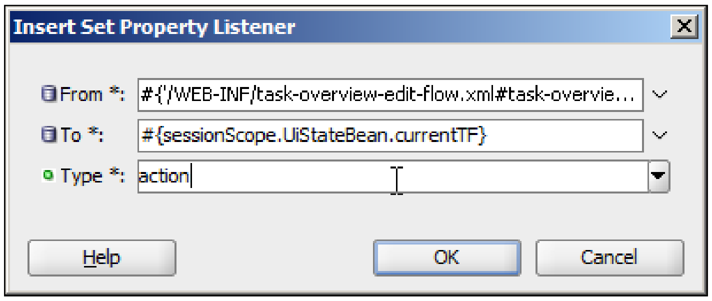
Component Palette: ADF Faces
ADF Faces
Common Components
Layout
Operations
Set Property Listener
Show Popup Behavior
Show Printable Page Behavior

Set Property Listener - Prope...

From *:	#{'/WEB-INF/task-overvi...
To *:	#{sessionScope.UiStateB...
Type *:	action

Inserting after Set Property Listener | Selected: Set Property Listener | Web Editing

In the **Insert Set Property Listener** dialog, set the **From** property to **#{'/WEB-INF/task-overview-edit-flow.xml#task-overview-edit-flow'}**, the **To** property to **#{sessionScope.UiStateBean.currentTF}** and **Type** to **action**:



From *: #{'/WEB-INF/task-overview-edit-flow.xml#task-overvie...
To *: #{sessionScope.UiStateBean.currentTF}
Type *: action

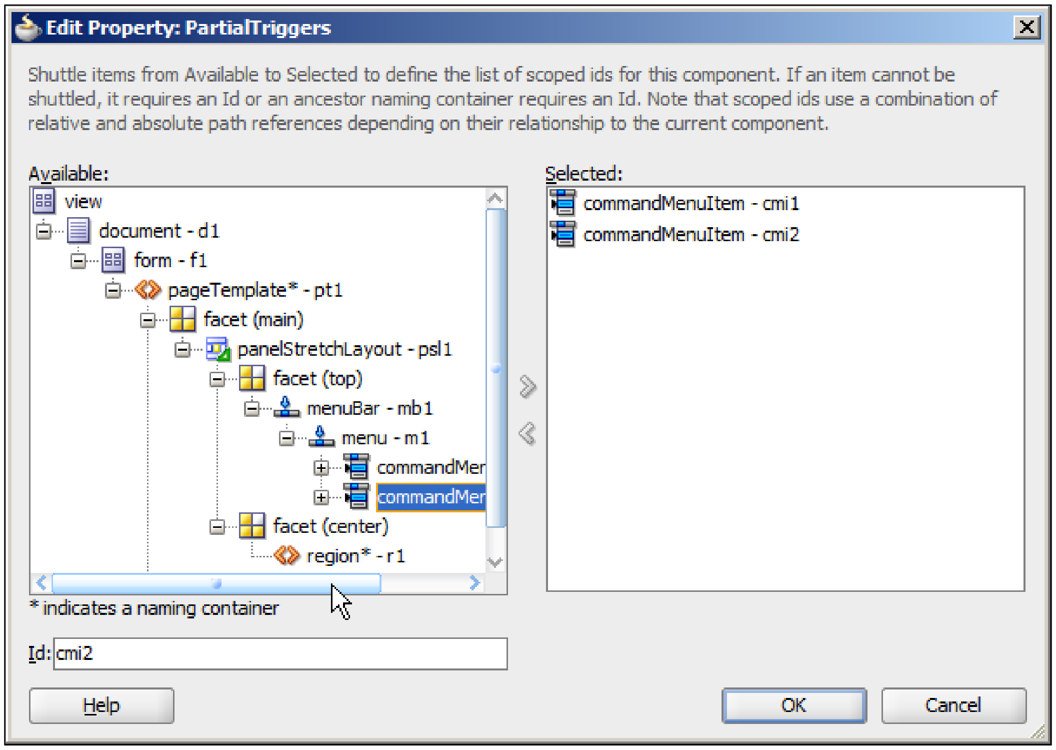
Repeat this for the **Timeline** command menu item, only set **From** to **#{'/WEB-INF/person-timeline-flow.xml#person-timeline-flow'}**.

These two property listener components will assign literal values (in the **From** fields) to the `currentTF` attribute in the `UiStateBean`. Again, this only serves as the simplest possible example of how to use a dynamic region—a real enterprise application would store the actual task flow URLs in some central repository (for example, a database table).

Making the region re-draw itself

The final part of the solution is a way to make the region redraw itself as needed. The ADF framework makes this very simple – all you need to do is to set the **PartialTriggers** property on the region component. The **PartialTriggers** property on a component is a list of all the other components that can trigger a refresh of that component.

Select the region and click on the little triangle next to the **PartialTriggers** property and choose **Edit**. This opens a dialog box where you can define the components that should trigger re-drawing of the region. Select the two **commandMenuItem** elements and move them to the right-hand side:



Edit Property: PartialTriggers

Shuttle items from Available to Selected to define the list of scoped ids for this component. If an item cannot be shuttled, it requires an Id or an ancestor naming container requires an Id. Note that scoped ids use a combination of relative and absolute path references depending on their relationship to the current component.

Available:

```
view
document - d1
  form - f1
    pageTemplate* - pt1
      facet (main)
        panelStretchLayout - psl1
          facet (top)
            menuBar - mb1
              menu - m1
                commandMer
                commandMer
          facet (center)
            region* - r1
```

* indicates a naming container

Id: cmi2

Selected:

commandMenuItem - cmi1
commandMenuItem - cmi2

In this way, whenever a user selects either of the menu items, the dynamic region component will re-draw itself, reflecting any change to the current task flow in the `UiState` bean.

Summary

In this chapter, you have built a Common Model workspace to supplement the Common Code and Common UI workspaces you built in *Chapter 5, Prepare to Build*. Using the ADF Libraries created from these three base workspaces, you have developed two subsystems, implementing the two use cases you first saw in *Chapter 1, The ADF Proof of Concept*.

You have used bounded task flows with page fragments to build well-defined, re-usable blocks of functionality that can easily be combined into a larger master application. You have seen how to use proper enterprise methodology, including task tracking to focus your work, and a central source repository.

You have deployed your subsystems as ADF Libraries and combined them in a master application, including both a menu and a dynamic region with all necessary supporting code to make your enterprise application look like a desktop application and not just a series of web pages. With this knowledge, you can build additional subsystems and add them to your master application until you have implemented all requirements.

One thing that we skipped over in this chapter, though, is the testing. That will be the subject of the next chapter.

7
Testing your Application

When a scientist after years of study finally discovers a promising chemical compound with the potential to cure cancer, does he crank up the pill-making machine right away? Of course not! He has already tested on cell cultures and lab animals for years, and an approved drug is still years away. Potential new drugs go through a rigorous and strictly regulated testing program starting with just a few humans, and leading up to large-scale Phase III clinical trials with hundreds of patients.

Your enterprise application is not likely to be subject to the same regulatory requirements, but that is no excuse for you to go from in vitro testing to full-scale deployment. You also need to test your application thoroughly through several phases and not simply dump it untested on your unsuspecting users.

The Holy Grail of software testing

The purpose of software testing is to ensure that the software meets requirements. Manual software testing can achieve this, but a manual test introduces a degree of variability into a process that should be strictly uniform; the tester might forget a test step, or might miss an answer deviating from the correct one.

That's why you should aim for as high a degree of **automated testing** as possible. If your test is run by a computer, it will be run the same way every time, and the computer won't miss small errors.

Initial tests

In the pharmaceutical business, a new drug candidate is first subjected to Phase I testing on a small number of healthy patients to test that the drug is safe for humans and works in the human body the way the models predict.

In your enterprise project, you similarly subject your code to initial, simple tests in order to ensure that each part of your code works the way you intend it to. These tests are often called **unit tests**, because they test the smallest possible unit of work. Good tools for this are **JUnit** (`http://junit.org`) and **TestNG** (`http://testng.org`).

> **Keep testing**
> Your unit tests must be stored together with the application source code so they can be run again after each change to the application. If requirements change, you should update the unit tests accordingly, and you should add additional unit tests when new functionality is added.

Working with JUnit

JUnit is a unit testing framework for Java, which means:

- It is used to test Java code
- The test cases are written in Java

Your test cases use annotations and some classes that are part of the JUnit framework, and you use other classes in the JUnit framework to actually run your tests.

With JUnit, you create **test classes** containing **test methods**. Inside these test methods, you place **assertions**—statements that *assert* what should happen. If your assertion is correct, the test is considered passed, otherwise it is considered failed.

There is a JUnit extension for JDeveloper to help you writing and running unit tests. For licensing reasons, JUnit is not delivered with JDeveloper, but you can easily install it using **Help | Check for Updates**. You find the JUnit extension under **Official Oracle Extensions and Updates**—choose both **BC4J JUnit Integration** and **JUnit Integration** (**BC4J**—**Business Components for Java** is the *old* name for ADF Business Components). You have to accept the JUnit license when installing this extension.

What to test with JUnit

JUnit excels at testing **code**—the classic example in the JUnit Cookbook (`http://junit.sourceforge.net/doc/cookbook/cookbook.htm`) tests a Java class for adding monetary amounts in multiple currencies.

If you have a class that does something simple like this, it is easy to write unit tests. But an ADF application is much more complicated, so it can be harder to identify what to test.

A good unit test

If you google "good unit tests", you will find many people offering ideas about what constitutes a good unit test. A good unit test probably has following characteristics:

- It tests one thing
- It can run in isolation
- It is easy to run
- It runs quickly

Your unit tests should test one unit of work—the smallest bit of code that it makes sense to test in isolation. A small unit tests points out exactly where the error is, while the failure of a big unit test just leads to more debugging.

Your unit tests should be able to run in isolation, in any order. To a unit testing purist, a unit test should not depend on external resources such as databases that is probably a bit unrealistic for an enterprise ADF application. Nevertheless, your unit tests should preferably set up their own test data so they can be run and re-run at any time and not depend on someone remembering to run a test-data-building script beforehand. That also allows the test to run as part of an automated build process.

Your unit tests should be easy to run; this is achieved by using the JUnit framework in JDeveloper, where you can simply run your test class and get feedback directly in JDeveloper by the *JUnit Test Runner*.

Your unit tests should run fast so you will actually run them and not be tempted to skip testing because it will hold up your work.

Unit testing ADF applications

It follows from these criteria that it is not easy to write a good unit test for your user interface. But that does not matter, we will leave user interface testing for later.

The functionality you should test during this phase is *all the Java code you have written*:

- If your framework extension classes have any content (we built them empty in *Chapter 5*, *Prepare to Build*, remember?), you need test cases to verify that they work as intended
- If your view objects or entity objects contain custom methods, you must test these
- If you have written and published custom methods in your application modules, you need to test these

Don't test ADF itself

You do not need to test the ADF framework itself; Oracle has already done that. This means that if you have defined an entity object on a table, and a view object on the entity object, you do not need to test that the view object actually retrieves the rows in the underlying table. This is a standard ADF functionality, and while testing it does not hurt, it does squander resources that could be used better.

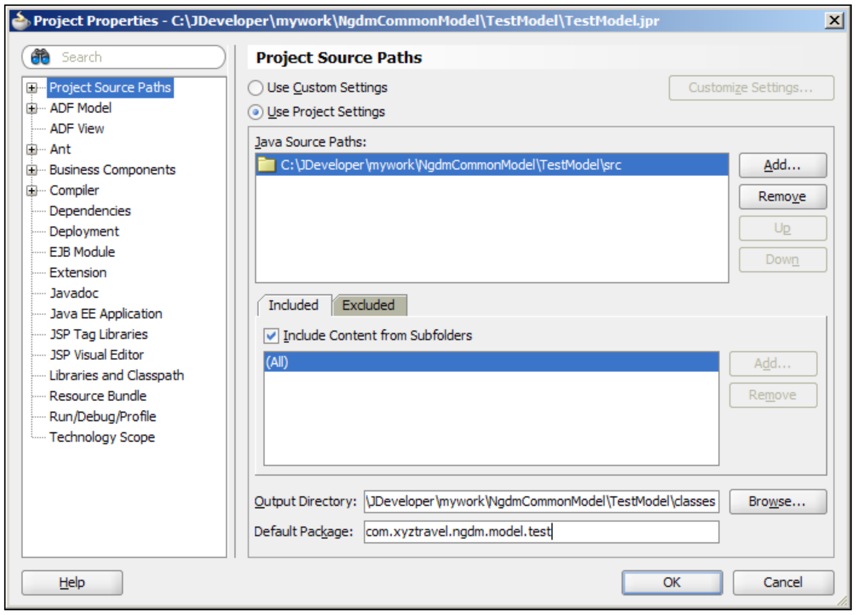
If you structure your code in a *common model* workspace and a number of *subsystem workspaces*, you will probably find that most of your unit tests are in the common model workspace, and some specialized view objects and application modules in the individual task flow workspaces are also unit tested.

Preparing for unit testing

As an example of how to set up unit testing, we will implement unit testing of the common model (in the `XdmCommonModel` workspace). We will keep our unit testing in a separate project in the workspace to avoid cluttering up the real common model workspace with test artifacts.

Setting up a test project

Open the workspace and choose **File | New** and then **Generic Project** (under **General**). Give the project the name `TestModel` and click **Finish** to create the project. You should define the default package name for Java code created in this project. To do this, right-click on the **TestModel** project and choose **Project Properties**, and then select **Project Source Paths** in the tree to the left. At the bottom-right of the dialog, set **Default Package** to something like `com.dmcsol.xdm.model`:



Project Source Paths

Use Custom Settings [✓ Use Project Settings]

Java Source Paths: C:\JDeveloper\mywork\NgdmCommonModel\TestModel\src

[✓] Include Content from Subfolders

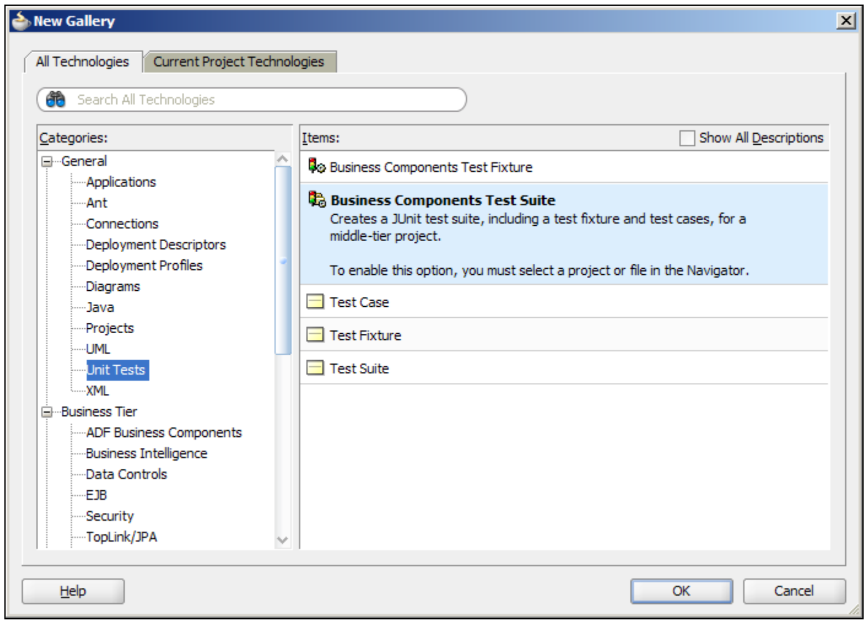
(All)

Output Directory: \JDeveloper\mywork\NgdmCommonModel\TestModel\classes

Default Package: com.xyztravel.ngdm.model.test

Adding default testing

Now select the **TestModel** project and choose **File** | **New** and then **General** | **Unit Tests** | **Business Components Test Suite**:



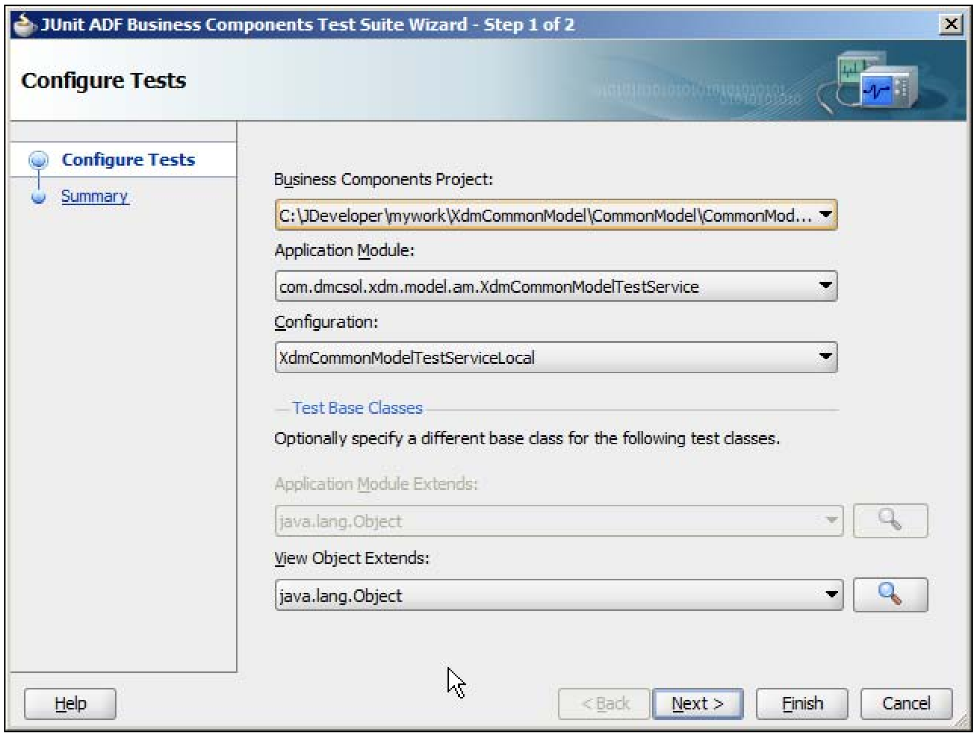
Business Components Test Suite

Creates a JUnit test suite, including a test fixture and test cases, for a middle-tier project.

To enable this option, you must select a project or file in the Navigator.

If you do not see the **Unit Tests** option, you probably did not install the **JUnit Integration** extension. If you see the **Unit Tests** option, but not **Business Components Test Suite**, you probably did not install the **BC4J JUnit Integration** extension.

The **JUnit ADF Business Components Test Suite Wizard** allows you to choose the business components project you wish to test, as well as the application module. Verify that the settings point to the application module you wish to test:



JUnit ADF Business Components Test Suite Wizard - Step 1 of 2

Configure Tests

Business Components Project: C:\JDeveloper\mywork\XdmCommonModel\CommonModel\CommonMod...

Application Module: com.dmcsol.xdm.model.am.XdmCommonModelTestService

Configuration: XdmCommonModelTestServiceLocal

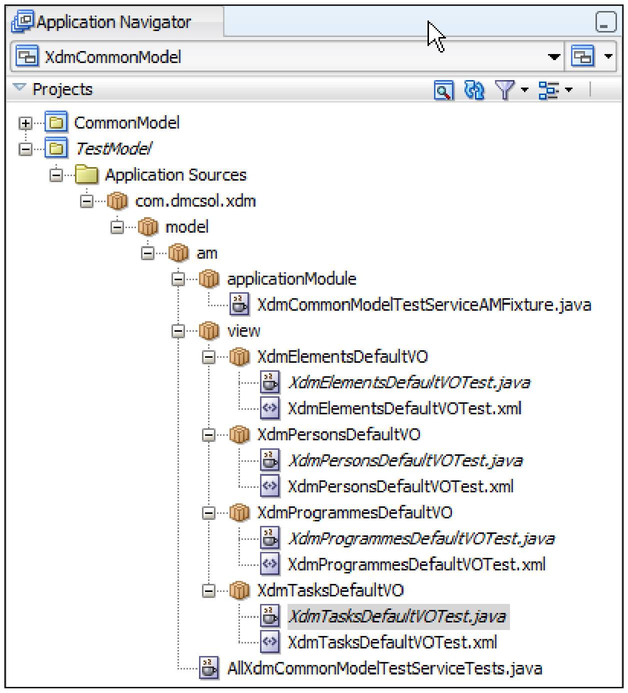
Test Base Classes

Optionally specify a different base class for the following test classes.

Application Module Extends: java.lang.Object

View Object Extends: java.lang.Object

Click **Finish** to complete the wizard. The **Application Navigator** panel will now show many new files:



```
CommonModel
TestModel
  Application Sources
    com.dmcsol.xdm
      model
        am
          applicationModule
            XdmCommonModelTestServiceAMFixture.java
          view
            XdmElementsDefaultVO
              XdmElementsDefaultVOTest.java
              XdmElementsDefaultVOTest.xml
            XdmPersonsDefaultVO
              XdmPersonsDefaultVOTest.java
              XdmPersonsDefaultVOTest.xml
            XdmProgrammesDefaultVO
              XdmProgrammesDefaultVOTest.java
              XdmProgrammesDefaultVOTest.xml
            XdmTasksDefaultVO
              XdmTasksDefaultVOTest.java
              XdmTasksDefaultVOTest.xml
          AllXdmCommonModelTestServiceTests.java
```

The wizard has created three kinds of objects for you:

- A **Test Fixture**
- A **Test Suite**
- A number of **Test Classes**

The *Test Fixture* takes care of creating an instance of the application module you wish to test. As this is an expensive operation, it makes sense to do this only once in your test run. You can open the `CommonModelTestServiceAMFixture.java` file to see what it does—it's standard code for programmatically accessing ADF application modules.

The *Test Suite* collects all the individual tests that make up your test suite. This class makes heavy use of Java **annotations**—the keywords prefixed with `@`. Annotations are a way to make metadata available for frameworks such as JUnit to use. For example, the `@Suite.SuiteClasses` annotation lists all the test classes that the suite must run.

```
@Suite.SuiteClasses( { XdmTasksDefaultVOTest.class,
                       XdmProgrammesDefaultVOTest.class,
                       XdmPersonsDefaultVOTest.class,
                       XdmElementsDefaultVOTest.class })
```

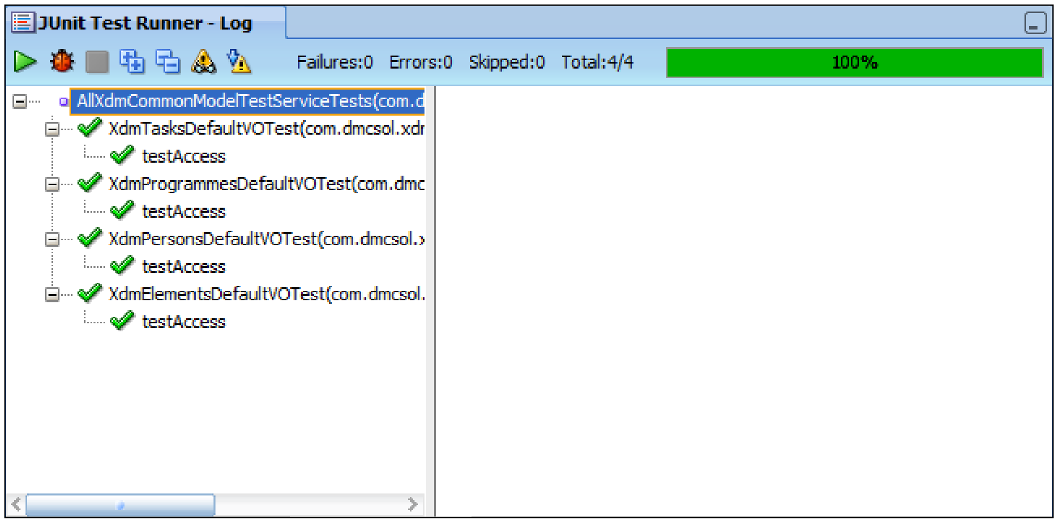
Finally, the framework has built a number of test classes—by default one for every view object included in the application module under test. These classes also use annotations to tell JUnit what each method does, and contain test methods to actually perform the test. The default test class generated for the `XdmTasksDefaultVO` view object looks like the following:

```
package com.dmcsol.xdm.model.am.view.XdmTasksDefaultVO;

import com.dmcsol.xdm.model.am.applicationModule.
XdmCommonModelTestServiceAMFixture;

import oracle.jbo.ViewObject;

import org.junit.*;
import static org.junit.Assert.*;

public class XdmTasksDefaultVOTest {
    private XdmCommonModelTestServiceAMFixture fixture1 =
XdmCommonModelTestServiceAMFixture.getInstance();

    public XdmTasksDefaultVOTest() {
    }

    @Test
    public void testAccess() {
      ViewObject view = fixture1.getApplicationModule().findViewObject
("XdmTasksDefault");
      assertNotNull(view);
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }
}
```

The *real* testing work happens in the test method or methods—those annotated with `@Test`. Each test method will *do* something and then *assert* that something is true. The preceding default test will get the application module under test (from the `fixture1` object that is retrieved from the test fixture class) and then find a specific view object (`XdmTasksDefaultVO`). It then asserts that the view object reference is not null, that is, the application module under test does indeed contain a `XdmTasksDefaultVO` view object instance. If this is the case, the test has succeeded. If this is not the case, the test fails.

To see the testing in action, right-click on the test suite class (**AllXdmCommonModelTestServiceTests.java**) and choose **Run**.

In the log window, you will see a new tab called **JUnit Test Runner** open. In this tab, you see the tests being executed and the results summarized:



```
AllXdmCommonModelTestServiceTests(com.d
  XdmTasksDefaultVOTest(com.dmcsol.xdr
    testAccess
  XdmProgrammesDefaultVOTest(com.dmc
    testAccess
  XdmPersonsDefaultVOTest(com.dmcsol.x
    testAccess
  XdmElementsDefaultVOTest(com.dmcsol.
    testAccess
```

You will see that four tests were executed (the default one test for each view object used in the application module), and they all succeeded (as expected). The progress bar is green to indicate that all tests passed.

Real unit testing example

Writing a test and watching it complete correctly does not really count—you might have an error in your test, which means you will always get a test success. You have not really proven that your test works *until you have seen it change state*.

This is the thinking behind the test-driven development where you write your tests first. As an example, we will implement the requirement that tasks are never actually deleted, but only marked cancelled. In the database, this is indicated by setting the `CXL_YN` column to the value `Y`.

Adding a test case

Because it must be possible to run the test in isolation, your test method cannot depend on data already in the database. In this case, we will let the test method itself add the record itself, but you might also create a set of test data in the test fixture class.

Keeping test data separate

If you insert dummy data into your tables, you should make sure that it is easily recognized so you can clean up any test data left over from a failed test method execution. One way of doing this is by using negative numbers for primary keys. Use large values such as -1001, -1002, and so on to avoid conflicts with the temporary keys ADF generates for DBDomain attribute values.

Your test method must do the following:

1. Add a new TASKS record (with a `-1001` ID).
2. Find the record through the view object, and call the `remove` method.
3. Verify that the record still exists, and that the `CxlYn` attribute now has the value `Y`.
4. Clean up: remove the TASKS record.

With a bit of cleanup, your `XdmTasksDefaultVO` class might look like the following:

```
package com.dmcsol.xdm.model.am.view.XdmTasksDefaultVO;
import …
public class XdmTasksDefaultVOTest {
private static CommonModelTestServiceAMFixture fixture1 =
CommonModelTestServiceAMFixture.getInstance();
private static ApplicationModule am =
fixture1.getApplicationModule();
public Tasks1VOTest() {
}
@Test
public void testAccess() {
ViewObject view = am.findViewObject("Tasks1");
assertNotNull(view);
}
@Test
public void testDelete() {
// add a test record directly to the table
Transaction tr = am.getTransaction();
tr.executeCommand("INSERT INTO tasks (task_id, start_date,
text) VALUES (-1, sysdate, 'Test Task')");)");
tr.commit();
// find the row with ID -1 and remove it
```

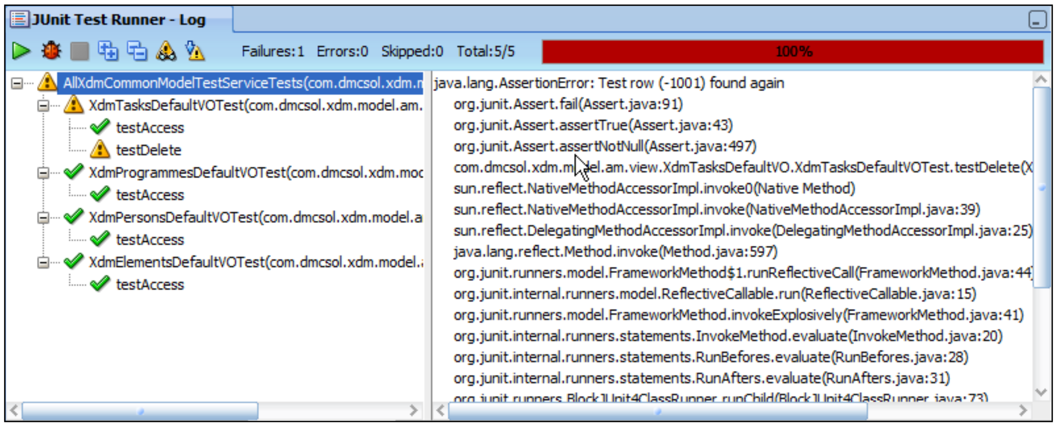
```
      ViewObject v = am.findViewObject("("Tasks1");
      Key k = new Key(new Object[] { -1 });
      Row r1 = v.getRow(k);
      assertNotNull("Test row (-1) found", r1);
      v.setCurrentRow(r1);
      v.removeCurrentRow();
      tr.commit();
      // look for the row
      v.executeQuery();
      Row r2 = v.getRow(k);
      assertNotNull("Test row (-1) found again", r2);
      // test that CxlYn attribute is now Y, indicating deletion
      assertEquals("Test row CxlYn value is Y", "Y",
        r2.getAttribute("CxlYn"));
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    @AfterClass
    public static void deleteTestData() {
      Transaction tr = am.getTransaction();
      tr.executeCommand("DELETE FROM tasks " +
        "WHERE task_id = -1");
      tr.commit();
    }
  }
```

The `testDelete()` method is new and performs the test precedingly described. It is annotated with `@Test` so that the JUnit framework knows to run it as a test case. The method uses `executeCommand()` to issue SQL directly to the connection the application module is using, and then uses `findViewObject()` to get a reference to the `XdmTasksDefaultVO` view object. An assertion checks that we did find the test row, and then we remove this row from the view object row set with `removeCurrentRow()`. It commits the transaction (to force execution of the logic in the entity object) and then re-executes the view object query to find the row again. The method asserts that the row is found, and that the `CxlYn` attribute has now been set to `Y`, indicating a logical delete.

An extra cleanup method `deleteTestData()` has also been added and given the annotation `@AfterClass`. This means that it will run as cleanup after all test methods in the class has been executed.

Finally, the fixture and the application module have been moved up as private static variables to simplify the code.

When you run the test suite, you will see the testDelete method fail with an `AssertionError` showing the text **Test row (-1001) found again**. This was the text we put into the `AssertNotNull` statement and is to be expected—after all, we have not implemented the logical delete functionality yet:



```
AllXdmCommonModelTestServiceTests(com.dmcsol.xdm.n
  XdmTasksDefaultVOTest(com.dmcsol.xdm.model.am.
    testAccess
    testDelete
  XdmProgrammesDefaultVOTest(com.dmcsol.xdm.moc
    testAccess
  XdmPersonsDefaultVOTest(com.dmcsol.xdm.model.a
    testAccess
  XdmElementsDefaultVOTest(com.dmcsol.xdm.model.
    testAccess
```

```
java.lang.AssertionError: Test row (-1001) found again
    org.junit.Assert.fail(Assert.java:91)
    org.junit.Assert.assertTrue(Assert.java:43)
    org.junit.Assert.assertNotNull(Assert.java:497)
    com.dmcsol.xdm.model.am.view.XdmTasksDefaultVO.XdmTasksDefaultVOTest.testDelete(X
    sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    java.lang.reflect.Method.invoke(Method.java:597)
    org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:44)
    org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:15)
    org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:41)
    org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:20)
    org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:28)
    org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:31)
    org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:73)
```

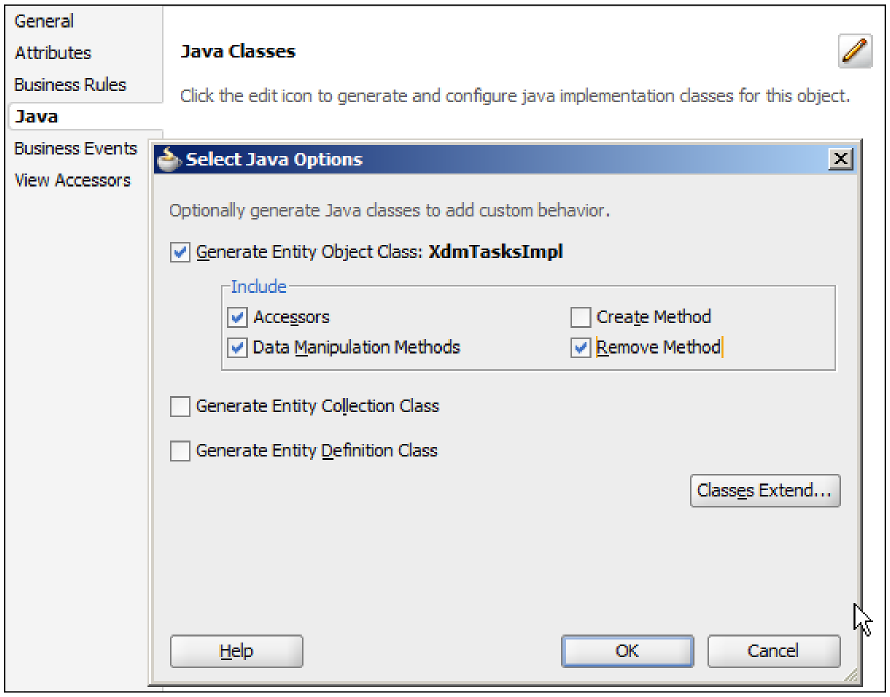
Implementing the logical delete

Implementing the logical delete—setting a marker on a record instead of actually deleting it requires two changes to the entity object:

- Changing the remove method to set the marker
- Changing the SQL statement from a DELETE to an UPDATE

This is not something that can be done in metadata, you need a Java class implementing your entity object.

Open the `XdmTasks` entity object and choose the **Java** sub-tab on the right. Then click on the little pencil icon at the top-right to generate Java code for this entity object. The **Select Java Options** dialog appears:



Java Classes

Click the edit icon to generate and configure java implementation classes for this object.

Select Java Options

Optionally generate Java classes to add custom behavior.

☑ Generate Entity Object Class: **XdmTasksImpl**

Include

☑ Accessors ☐ Create Method
☑ Data Manipulation Methods ☑ Remove Method

☐ Generate Entity Collection Class

☐ Generate Entity Definition Class

Choose to generate methods for **Accessors**, **Data Manipulation Methods**, and **Remove Method**. It does not hurt to select the **Create Method** as well, or you can override this method later if you need it. When you click **OK**, an `XdmTasksImpl.java` file is created and can be seen in the **Application Navigator** under the `XdmTasks` entity object node.

In this class, first find the `remove()` method. It is probably easiest to find the method in the **Structure Panel** at the bottom-left, where all the methods are listed in alphabetical order. Double-click on the `remove()` method to jump to that method in the main editing window. Add a `setCxlYn()` instruction so your `remove()` method looks like the following:

```
public void remove() {
  // set cancel flag. doDML() will change DELETE to UPDATE
  setCxlYn("Y");
  super.remove();
}
```

This means the `CxlYn` attribute (corresponding to the `CXL_YN` column in the database) must be set to `Y`, and then the normal `remove()` processing should continue.

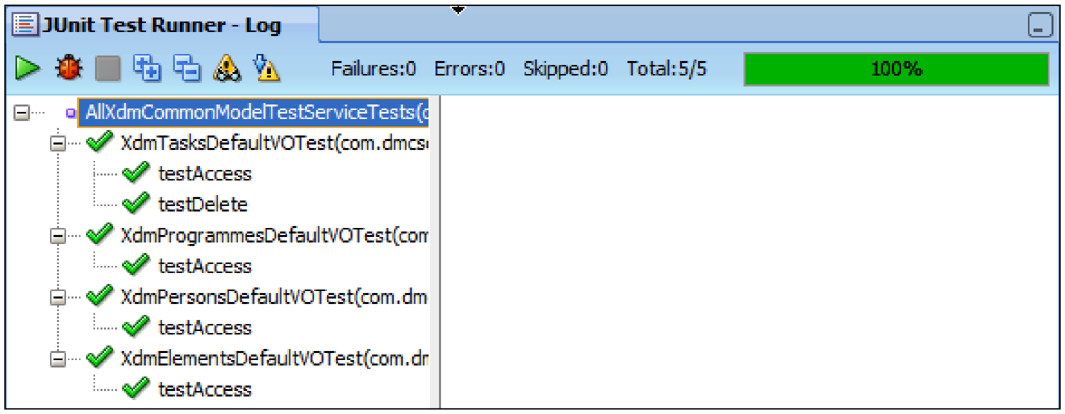
Then jump to the `doDML()` method. This method is invoked whenever the framework needs to execute a **Data Manipulation Language** (**DML**) SQL statement against the database—INSERT, UPDATE, or DELETE. This means that it is invoked after the preceding `remove()` method to actually perform the operation against the database. Change this method to look like the following:

```
protected void doDML(int operation, TransactionEvent e) {
  if (operation == DML_DELETE) {
    super.doDML(DML_UPDATE, e);
  } else {
    super.doDML(operation, e);
  }
}
```

This code will simply intercept a DELETE and do the normal processing associated with UPDATE. As the `remove()` method already sets the marker to `Y`, this is all we need.

Re-testing

Now you can go back to the *TestModel* project and run the test suite (**AllXdmCommonModelTestServiceTests**) again. You should now see the progress bar turn green, indicating that all five tests passed:



```
Failures:0  Errors:0  Skipped:0  Total:5/5  100%

AllXdmCommonModelTestServiceTests(c
  XdmTasksDefaultVOTest(com.dmcs
    testAccess
    testDelete
  XdmProgrammesDefaultVOTest(com
    testAccess
  XdmPersonsDefaultVOTest(com.dm
    testAccess
  XdmElementsDefaultVOTest(com.dr
    testAccess
```

Automating unit testing

It is, of course, great that the developer has a test suite that he or she can run against the code at all times but the full value of having a collection of unit tests comes if you integrate them with an automated build process. We will discuss automated build in *Chapter 11, Package and Deliver*.

User interface tests

After Phase I testing of a drug candidate has successfully established that it is safe to use, Phase II testing commences. This is performed on larger groups of patients to test that the drug is actually effective against the ailment that it is supposed to cure.

In your enterprise project, your next testing phase is a test of the user interface, testing that your application actually meets the user requirements. One tool for verifying this is **Selenium**.

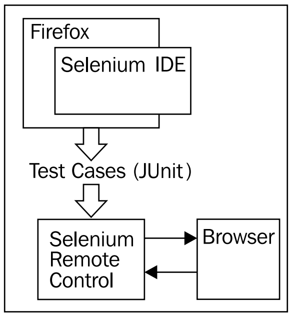
Working with Selenium

Selenium is an open source tool for testing web applications, even web application that make heavy use of JavaScript, which ADF does. It works by:

- Recording a user session in a browser, including assertions about what is expected to happen
- Exporting the recorded session to a programming language (we will use Java)
- Playing back the recorded session and comparing the results to the recorded assertions

Selenium consists of two parts:

- The Selenium IDE is a Firefox plug-in where you record your test session. From the IDE, you can play back your session to quickly check that it works, and export it as a Java JUnit test case
- Selenium Remote Control Server is a Java-based command line server that will execute your recorded test using a browser



Between the recording and the playback, you can modify the test in JDeveloper. This allows you to add parameters and move beyond simple record/playback testing.

What to test with Selenium

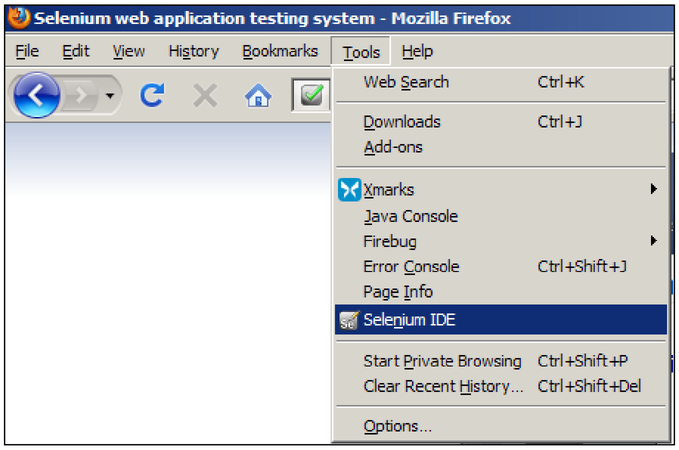
Selenium tests web pages, so you should have at least one test for every non-trivial page in your application. If your requirements are formulated as use cases, they will typically contain all the steps you need to go through in your user interface test. If not, you will have to start by writing out a list of the steps the user would go through when using your application, so you know what to record.

A Selenium test will compare the content of the web page to the recorded assertions. For example, you could assert that a specific text is part of the page, or that a field has a specific value. Generally, a Selenium test does not test the look of the application—if you change the look of the application, for example through skinning as explained in *Chapter 8*, *Look and Feel*, a Selenium test case will still give the same result.

Again, do not test the ADF framework itself—if you have built a simple view object on a single entity object, and dropped it onto a page as an ADF table without modifying it, you can trust ADF to take care of the details.

Installing Selenium

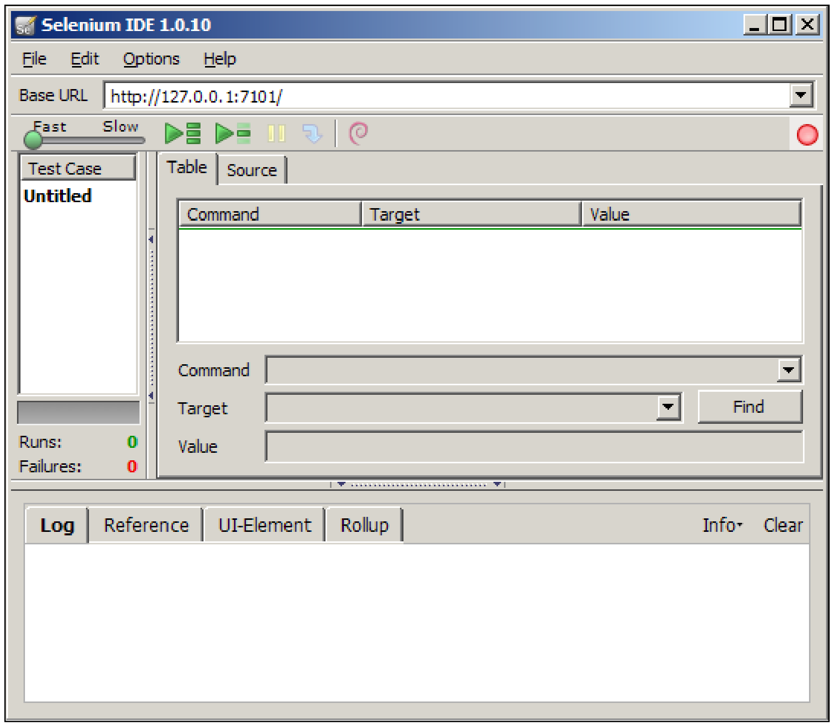
The Selenium IDE only exists as a Firefox plug-in, so you need the Firefox browser. If you do not already have it, you can download it from `http://www.firefox.com`. The Selenium IDE can be downloaded from within Firefox under **Tools** | **Add-ons** or you can go to `http://seleniumhq.org` to download. When you have accepted to install the Selenium add-on and restarted Firefox, you'll see the **Selenium IDE** menu item on the **Tools** menu in Firefox:


Selenium web application testing system - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
Web Search Ctrl+K
Downloads Ctrl+J
Add-ons
Xmarks
Java Console
Firebug
Error Console Ctrl+Shift+J
Page Info
Selenium IDE
Start Private Browsing Ctrl+Shift+P
Clear Recent History... Ctrl+Shift+Del
Options...

A simple test with Selenium

To record a test session with Selenium, you first need to start the application. For this example, we will test the Task Overview and Edit page task flow from the `XdmUC008` workspace. Open this workspace and then the `UC008View` project. Right-click on the **TestTaskOverviewEdit.jspx** page (under **Web Content**, **testpages**) and choose **Run**. The application starts and shows up your web browser.

Then choose the **Selenium IDE** menu item from the **Tools** menu in Firefox. The **Selenium IDE** window appears:



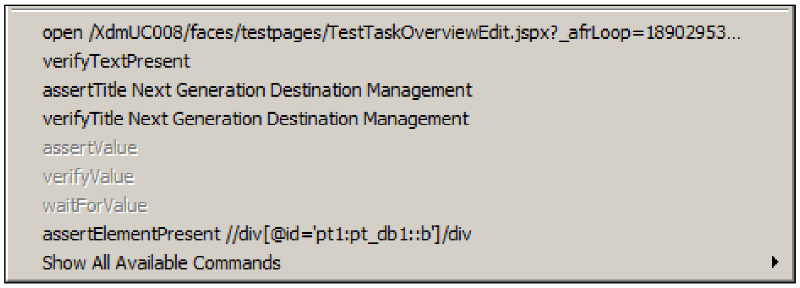
By default, it is already recording, shown by the light gray box around the red recording button in the right side of the Selenium IDE window toolbar.

Go back to the browser address line and copy the page URL *without* parameters. You will probably have noticed that ADF applications have fairly long URLs—everything to the left of the question mark sign is the real URL of the page, and everything to the right are session-specific parameters that the ADF framework has added. You just want the real URL, something like `http://127.0.0.1:7101/XdmUC008/faces/TestTaskOverviewEdit.jspx`.

Shortening the URL

To shorten the long default JDeveloper URL, go to the project properties of the `UC008View` project and choose the **Java EE Application** node. Change both the **Java EE Web Application Name** and **Java EE Web Context Root** fields to just the name of your workspace, for example, `XdmUC008`.

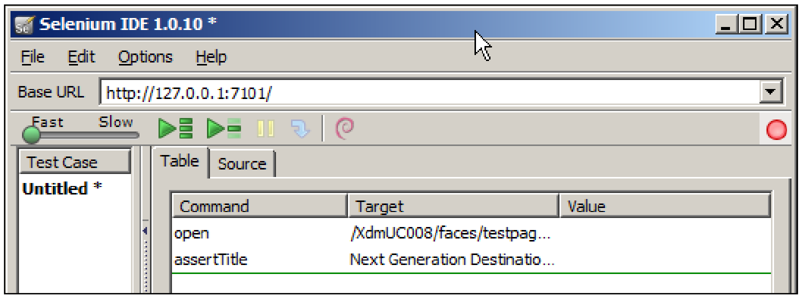
Clear the browser address bar and paste in the application URL. The page loads in the browser, but there is nothing to see in the Selenium IDE window. However, if you right-click anywhere on the page, you will see that the context menu in the browser has acquired a number of new options. The options will depend on where you click—if you click on the page background (outside any texts or fields), you will see something like the following:


open /XdmUC008/faces/testpages/TestTaskOverviewEdit.jspx?_afrLoop=18902953...
verifyTextPresent
assertTitle Next Generation Destination Management
verifyTitle Next Generation Destination Management
assertValue
verifyValue
waitForValue
assertElementPresent //div[@id='pt1:pt_db1::b']/div
Show All Available Commands

Choose the **assertTitle...** option (for example, **assertTitle Next Generation Destination Management**). This is a testing step, requiring that the title of the web page has a specific value.

The title of the web page is set through the **Title** property of the `af:document` tag on a page.

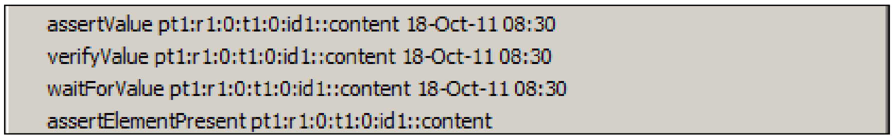
You should now see two commands registered in Selenium: An **open** command and the **assertTitle** you just selected:



Command	Target	Value
open	/XdmUC008/faces/testpag...	
assertTitle	Next Generation Destinatio...	

You can continue to use the application and let Selenium record your actions. If for example you make a selection from the **Responsible** drop-down box and click on the **Search** button, you will see Selenium register a **select** command and two **click** commands.

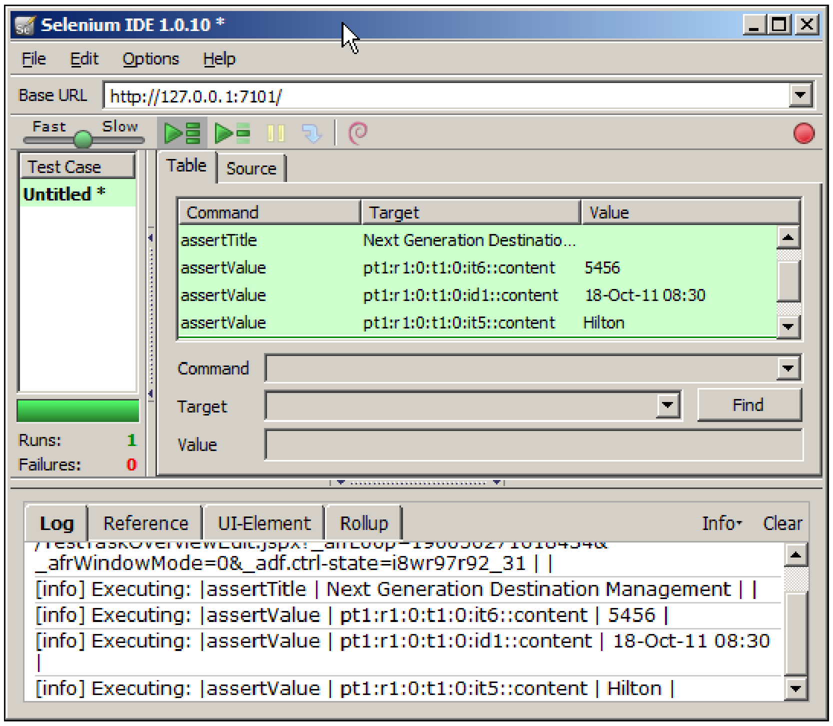
If you right-click on a specific field, you will see a number of options related to that specific field:



```
assertValue pt1:r1:0:t1:0:id1::content 18-Oct-11 08:30
verifyValue pt1:r1:0:t1:0:id1::content 18-Oct-11 08:30
waitForValue pt1:r1:0:t1:0:id1::content 18-Oct-11 08:30
assertElementPresent pt1:r1:0:t1:0:id1::content
```

Normally, you will use `assertValue` to assert that the field has a specific value (we'll get back to these options later).

When you are happy with your test, click the red recording button in the Selenium IDE window to stop recording. For now, move the speed slider in the left side of the Selenium IDE toolbar to the middle to avoid running your test faster than ADF can handle—we will address a few of the challenges in a later section. Then click one of the green **Play** buttons to run your test:



Base URL: http://127.0.0.1:7101/

Untitled *

Command	Target	Value
assertTitle	Next Generation Destinatio...	
assertValue	pt1:r1:0:t1:0:it6::content	5456
assertValue	pt1:r1:0:t1:0:id1::content	18-Oct-11 08:30
assertValue	pt1:r1:0:t1:0:it5::content	Hilton

Runs: 1
Failures: 0

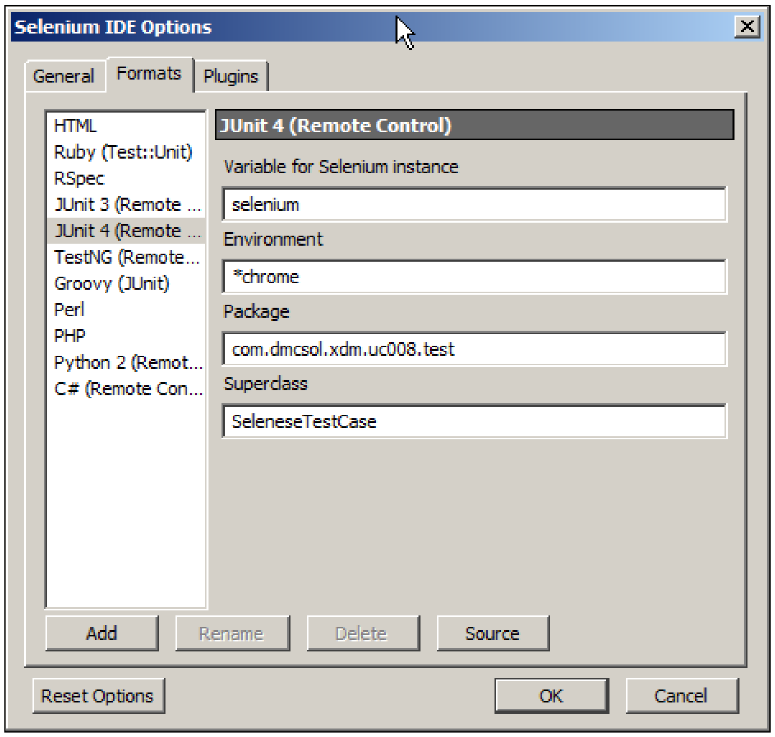
```
_afrWindowMode=0&_adf.ctrl-state=i8wr97r92_31 | |
[info] Executing: |assertTitle | Next Generation Destination Management | |
[info] Executing: |assertValue | pt1:r1:0:t1:0:it6::content | 5456 |
[info] Executing: |assertValue | pt1:r1:0:t1:0:id1::content | 18-Oct-11 08:30
|
[info] Executing: |assertValue | pt1:r1:0:t1:0:it5::content | Hilton |
```

Automating user interface tests

It is fine to be able to run your user interface tests automatically from a browser, which greatly reduces the risk of faulty testing due to human error. But we would like more: we want to be able to run the test automatically as part of our build process.

Fortunately, Selenium allows us to export our test cases as JUnit tests written in Java—just like the JUnit tests we write ourselves for our business components.

Before you export your test case in JUnit format, choose **Options** | **Options**, the **Formats** tab and **JUnit 4 (Remote Control)** in the left-hand box. In the right-hand side, set the package name to a test package in your project, for example **com.dmcsol.xdm.uc008.test**:



JUnit 4 (Remote Control)

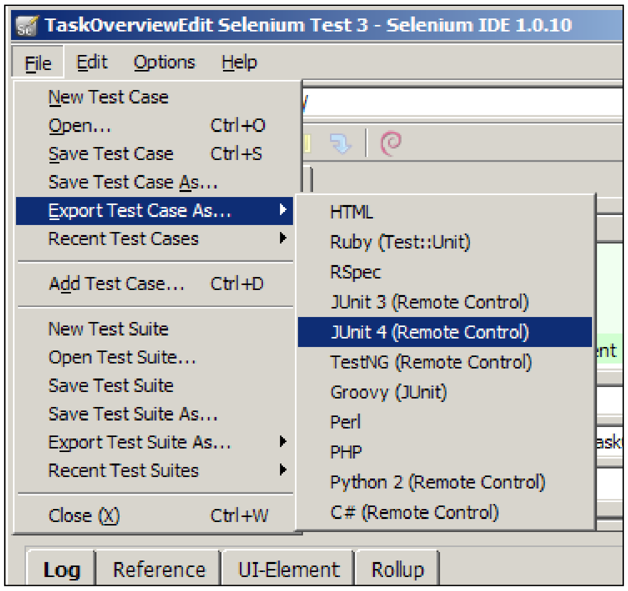
Variable for Selenium instance: selenium

Environment: *chrome

Package: com.dmcsol.xdm.uc008.test

Superclass: SeleneseTestCase

Then choose **File** | **Export Test Case As ...** | **JUnit 4 (Remote Control)**:



Give your test case a name (including a `.java` extension, for example, `UC008InitialData.java`) and save it. Do not use spaces in the name, because the name will be used as class name inside the file. If you open the Java file, you will see that it is normal Java with annotations just like your JUnit classes. It might look something like the following:

```
package com.dmcsol.xdm.uc008.test;

import com.thoughtworks.selenium.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import java.util.regex.Pattern;

public class UC008InitialData
extends SeleneseTestCase {
@Before
public void setUp() throws Exception {
selenium = new DefaultSelenium("localhost", 4444,
"*chrome", "http://127.0.0.1:7101/");
selenium.start();
}
```

```
    @Test
    public void testUC008InitialData() throws Exception {
      selenium.open("/XdmUC008/faces/testpages/TestTaskOverviewEdit.
jspx?_afrLoop=190056271618434&_afrWindowMode=0&_adf.ctrl-
state=i8wr97r92_31");
      assertEquals("Next Generation Destination Management", selenium.
getTitle());
      assertEquals("5456", selenium.getValue("pt1:r1:0:t1:0:it6::conte
nt"));
      assertEquals("18-Oct-11 08:30", selenium.getValue("pt1:r1:0:t1:0:i
d1::content"));
      assertEquals("Hilton", selenium.getValue("pt1:r1:0:t1:0:it5::cont
ent"));
    }
    }

    @After
    public void tearDown() throws Exception {
      selenium.stop();
    }
  }
```

Setting up to run Selenium JUnit tests

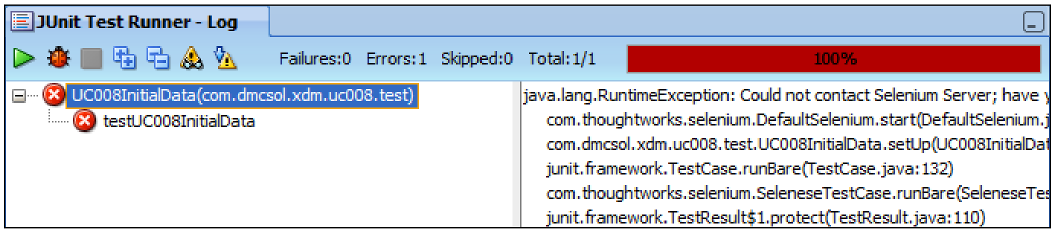
As you can see from the preceding code example, the Selenium JUnit tests make use of some `com.thoughtworks.selenium` classes that you need to make available. Your build/configuration manager should download the Selenium server from `http://seleniumhq.org/download`, place it in the common library directory, add it to your source control system, and commit.

Each developer should then get the file from the source control system. For the purpose of the example in this chapter, you can assume the role of build/configuration manager and place the JAR file (at the time of writing, it was called `selenium-server-standalone-2.0b3.jar`) in your common library directory, for example, `C:\JDeveloper\XdmLib`.

Then create a new project in your task workspace to hold your tests—just like you created a separate project for the JUnit tests for the business components. Choose **File | New** and then **Generic Project**. Give your project a name (for example, `UC008Test`) and click **Finish** to create the project. Under project properties, set the **Default Package** to the same as the base name for the task flow (for example, `com.dmcsol.xdm.uc008`). Also under project properties, go to the **Libraries and Classpath** node, click **Add JAR/Directory** and point to the Selenium RC JAR file in your common library directory. Additionally, click **Add Library** and add the **JUnit 4 Runtime** library.

Now, you need to place the actual Selenium test class into the project. It is easier to simply create a new class with the right name and package inside the project and then cut and paste the actual java code in. Use **File** | **New**, **Java Class**, give the same class name as you used when you exported it (`UC008InitialData`) and the same package (`com.dmcsol.xdm.uc008.test`). Then cut and paste in the content of the `.java` file you created from the Selenium IDE.

You are actually able to run your test now by right-clicking on the class and choosing **Run**. Because the code is a valid JUnit test case, JDeveloper recognizes it and tries to run it with the JUnit Test Runner:


JUnit Test Runner - Log
Failures:0 Errors:1 Skipped:0 Total:1/1
100%
UC008InitialData(com.dmcsol.xdm.uc008.test)
testUC008InitialData

```
java.lang.RuntimeException: Could not contact Selenium Server; have y
    com.thoughtworks.selenium.DefaultSelenium.start(DefaultSelenium.j
    com.dmcsol.xdm.uc008.test.UC008InitialData.setUp(UC008InitialDat
    junit.framework.TestCase.runBare(TestCase.java:132)
    com.thoughtworks.selenium.SeleneseTestCase.runBare(SeleneseTes
    junit.framework.TestResult$1.protect(TestResult.java:110)
```

The reason it fails is of course that we have not yet started the Selenium Server that actually executes the test cases and invoke the browser.

Starting the Selenium server

As you might remember from the introduction to Selenium, the Selenium IDE records the test cases inside a browser, and the Selenium Remote Control Server plays them back, using an external browser. When you run your JUnit Selenium test cases, they contact the Selenium server with instructions, and the server invokes a browser.

To start the Selenium server, open a command prompt and go to the directory where you placed the Selenium server JAR file (for example, `C:\JDeveloper\XdmLib`). Issue the following command:

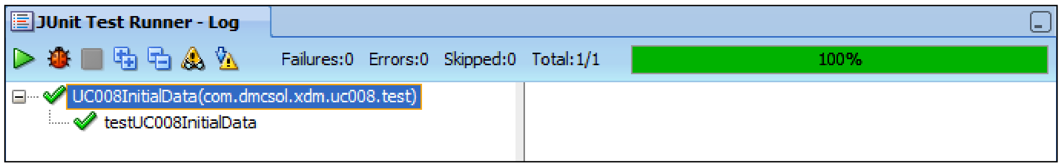
```
java -jar selenium-server-standalone-2.0b3.jar
```

Run the JAR file you downloaded—the preceding name was valid at the time of writing, but there is probably another version of the Selenium server out by the time you read this.

The server will start on the default port of 4444, and your JUnit test case initializes the `selenium` object with this port, so you are ready to run.

Running the test

You can simply right-click on your test case and run it again. You should see Selenium start up the browser, run through the test, close the browser and report the result back to JDeveloper:



Failures:0 Errors:0 Skipped:0 Total:1/1 100%

UC008InitialData(com.dmcsol.xdm.uc008.test)
testUC008InitialData

You might have noticed that the `DefaultSelenium()` call by default uses `*chrome` as browser (third parameter). If you do not have Google Chrome installed, it will run your default browser. You can also change this to, for example, `*firefox` or `*iexplore` to run the test in a different browser. In general, it is enough to test your application in one browser—you are using ADF components supplied by Oracle, and Oracle continually works to make sure these components work correctly across browsers.

Because the tests are JUnit tests, it is possible to integrate them into your build process like the business component JUnit tests. If you want to include Selenium tests in your automated process, you must change to generated code to take the URL of the application to test as an input parameter, and you need to make sure your process starts the application and the Selenium Remote Control Server before the UI test is run.

Using Selenium effectively

While a simple test like the one previously shown is fairly easy to run, Selenium is a powerful tool with many options and ADF pushes the web browser to the limit, using massive amounts of JavaScript. You will need to read at least some of the Selenium documentation at `http://seleniumhq.org/docs` if you want to use Selenium to test your enterprise project.

Value checking options

Much of your application testing will concern values inside components, for example, input fields. When you right-click on a field while recording, you get four options:

- **assertValue**: It will record an absolute demand that the value is correct. If the field in the browser does not have the right value when you run the test, the test will be aborted.

- **verifyValue**: It will record a requirement for a specific value. If the field in the browser does not have the right value when the test is played back, it will be registered as a failure, but the test will still continue to next test step.
- **waitForValue**: It will record a requirement for a specific value to appear before a configurable timeout. If you know the application might take several seconds to respond, you use a **waitForValue**. If the value appears in the field before timeout, the test continues. If the timeout appears, the test is a failure. You can configure a timeout in the IDE under **Options | Options**. The default is very high (30 seconds).
- **assertElementPresent**: It will record a requirement for the existence of a specific element, but does not check the value in the field. As long as a field with the recorded ID exists on the screen, the test is considered passed.

Lazy content delivery

When you navigate between records and assert values in an ADF application, you might experience that your Selenium test fails for no good reason. This is because ADF issues a request for new data asynchronously (without refreshing the whole page), and Selenium does not know that it must wait a fraction of a second for the data.

Additionally, some ADF components (such as, `af:table`) have a `ContentDelivery` property that can be set to `Lazy`. This means that the page will render quickly, and the table data will be delivered in a second request to the application server. This causes the same kind of confusion where Selenium notices that the page is loaded and starts running the assertions, while table data is still coming in.

You can mitigate these problems somewhat by using `waitForValue` (for navigation between records) and `waitForElementPresent` (for lazy-loading components). Of course, you also need to set your timeout to a reasonable value so you will not be waiting 30 seconds before detecting a failure.

Testing context menus

The Selenium IDE uses the right-click (context) menu to record test steps but if your application also uses a context menu, the browser context menu will never be shown. This means that you have to manually enter the test steps for context menus in the Selenium IDE window.

To test the context menu of your application itself, you use the `contextMenuAt` command in Selenium—you can look up the details in the Selenium documentation.

Verifying item ID

If your application is going to be customized (now or sometime in the future), you should use `assertElementPresent` to verify the presence of all user interface elements. This command takes an element ID as parameter and testing for ID will catch the situation where an `inputText` has changed ID from `it11` to `it12`.

The application will work with different component IDs, but if anybody has customized the application, the customization is tied to the component ID. Therefore, if you change the ID of a component, you break any customization made of that component. We will return to customization in *Chapter 9*, *Customizing Functionality*.

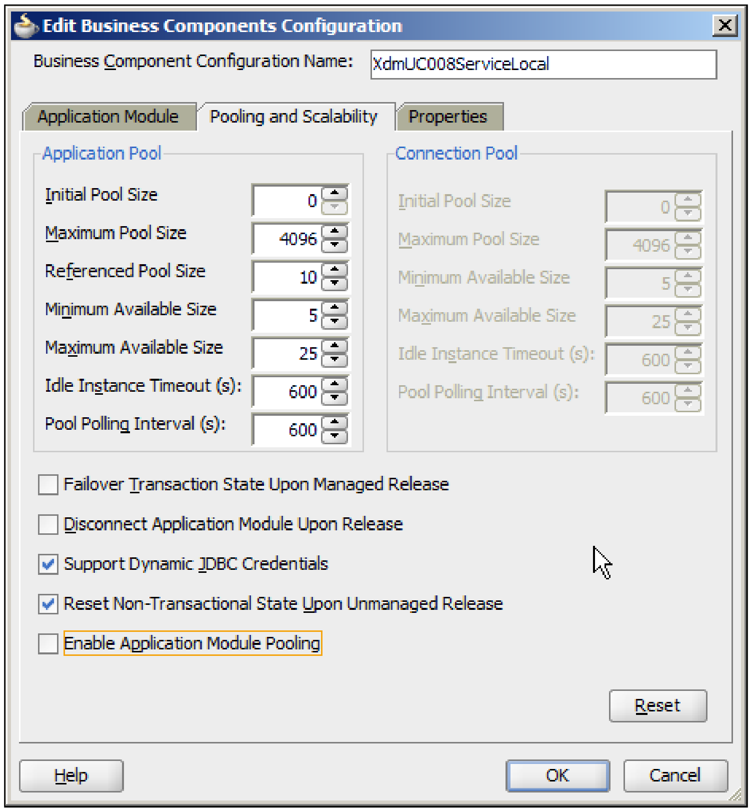
Testing passivation and activation

The user interface tests you have recorded and played back have only been simulating one user, and in the following section, we will simulate many. But there is one part of ADF testing that requires a bit of extra care: application module **passivation** and **activation**.

For performance reasons, ADF keeps a pool of application modules in memory. It tries to give each session the same application module as the session used during the last request; however, this might not be possible during peak load of your application. In this case, ADF saves the application modules state in a database table so the application module can be used by another session. This is called passivation. When the first session needs the application module again, its state is retrieved from the database—a process know as activation.

If you have made an error in your code and depend on some variable that is not persisted correctly when your application module state is stored, you will experience mysterious errors under high load.

To make sure you do not hit this problem, right-click on your application module, choose **Configurations**. By default, each application module has two configurations. Ensure that the one ending in …`Local` is selected and then click **Edit**. Then choose the **Pooling and Scalability** tab and *deselect* the checkbox **Enable Application Module Pooling**:



Business Component Configuration Name: XdmUC008ServiceLocal

Application Pool

Initial Pool Size	0
Maximum Pool Size	4096
Referenced Pool Size	10
Minimum Available Size	5
Maximum Available Size	25
Idle Instance Timeout (s):	600
Pool Polling Interval (s):	600

Connection Pool

Initial Pool Size	0
Maximum Pool Size	4096
Minimum Available Size	5
Maximum Available Size	25
Idle Instance Timeout (s):	600
Pool Polling Interval (s):	600

- [ ] Failover Transaction State Upon Managed Release
- [ ] Disconnect Application Module Upon Release
- [✓] Support Dynamic JDBC Credentials
- [✓] Reset Non-Transactional State Upon Unmanaged Release
- [ ] Enable Application Module Pooling

This forces the ADF framework to always store application module state in the database between requests. Run your user interface tests again to make sure none of these tricky persistence bugs lurk in your code. They will typically show up as `NullPointerExceptions` when an object you believed had a value is suddenly empty, because the application module has been passivated and re-activated.

Remember to enable application module pooling again before deploying the application for stress/performance testing.

ADF tuning

There are many ways of tuning the application module pool and other aspects of the ADF framework. Refer to the *Oracle Fusion Middleware Performance and Tuning Guide*, which has a chapter on tuning ADF. There is also a chapter on tuning application module pools in the *Fusion Developers Guide for Oracle Application Development Framework*.

Stress/performance tests

When a new drug candidate has successfully completed Phase II testing and has proven that it works, it moves on to Phase III testing with even more people to verify that it works on a larger scale.

In your enterprise project, successfully completing user interface testing proves that the application works for an individual user—now you need to prove that it is robust enough to handle real-life load. To do this, you can use another open source tool: **JMeter**.

Working with JMeter

JMeter is a tool for load testing web applications. Like Selenium, it records a user session and plays it back, but it does not attempt to really run the user session in the browser (like Selenium does)—it simply sends off the requests that a browser would make to the application server and measures the time it takes for the application server to respond.

Because it does not invoke a browser, but simply sends requests and receives responses, one workstation with JMeter can simulate the load of dozens or even hundreds of real-time users.

What to test with JMeter

At this point in your testing, you already know that your application works the way it's supposed to. What remains is:

- To test that the application works with multiple concurrent users
- To test how many users your system can handle

The majority of the errors you will find during this phase of testing will be **concurrency** issues—strange things that happen when more than one user is using your application. These can slip through both the programmer's own initial tests, unit tests, and user interface tests, because all of these only run one session.

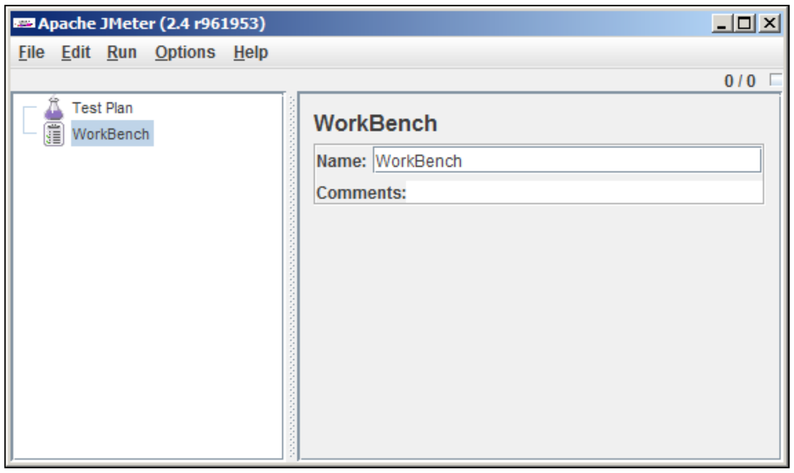
Your initial stress/performance tests should be run with just a small number of simulated concurrent users—five concurrent sessions are normally enough to tease out any concurrency issues in your application.

Once you are sure your application works with multiple users, you can scale up the load to ensure that your application meets the performance requirements.

Installing and running JMeter

You can download JMeter from `http://jakarta.apache.org/jmeter`. It is a Java application and requires Java 1.5 or later installed on your workstation.

To start JMeter, you simply execute the `jmeter.bat` file (on Windows) or `jmeter` (on Linux). The JMeter main window appears:



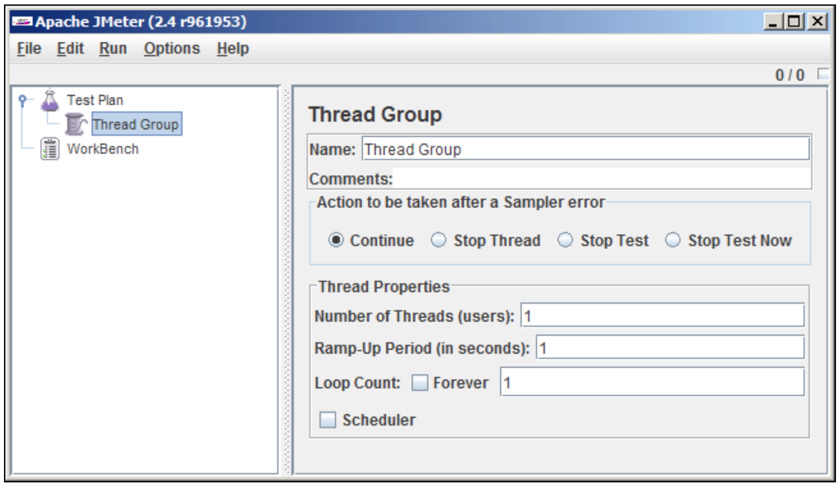
WorkBench

Name: WorkBench

Comments:

A simple test with JMeter

A JMeter test plan starts with a **Thread Group**, which defines the number of concurrent processes you will let loose on your application. To add a thread group, right-click the **Test Plan** node and choose **Add | Threads (Users) | Thread Group**:



Thread Group

Name: Thread Group

Comments:

Action to be taken after a Sampler error

[✓ Continue] Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 1

Ramp-Up Period (in seconds): 1

Loop Count: ☐ Forever 1

☐ Scheduler

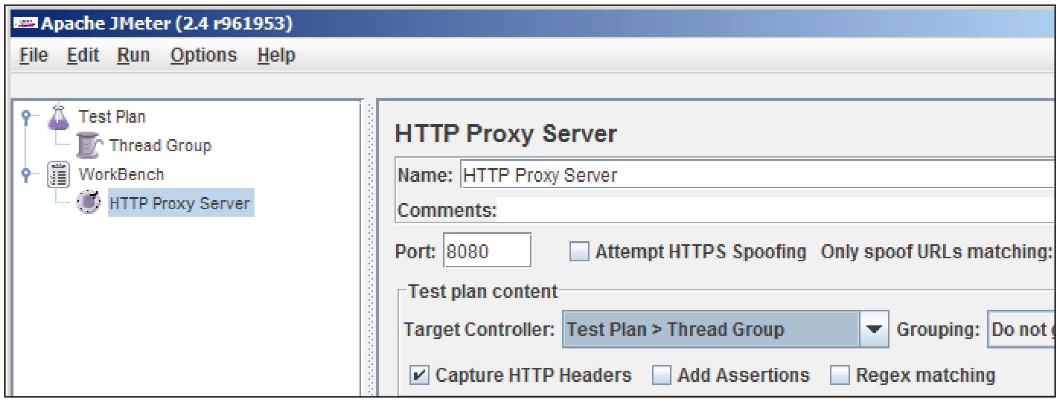
Here you can define the number of users, how many repetitions of your test, and so on. While defining and testing your JMeter test script, you should use only one user.

Once you have defined the thread group, you can start adding test elements manually by right-clicking on your thread group and choosing **Add**. However, like in Selenium, there is an easier way: recording the test.

Setting up JMeter as a proxy

To record a test, you set up JMeter as a proxy—a kind of gateway that all your web requests go through. If you work in a large corporation behind a firewall, your Internet access is likely to go through a proxy that will filter out some sites, protect against viruses, and so on. The JMeter proxy doesn't do any of this—what it does is to record every request for later playback.

To set up JMeter as a proxy, right-click the **Workbench** node and choose **Add | Non-Test Elements | HTTP Proxy Server**. Enter a port that is not already in use (for example, `8080`) and set the **Target Controller** to your thread group. If you wish, you can exclude certain URL patterns from the capture that JMeter does. This is useful, for example, if you have browser plug-ins that might contact some server in the middle of your recording:



HTTP Proxy Server

Name: HTTP Proxy Server

Comments:

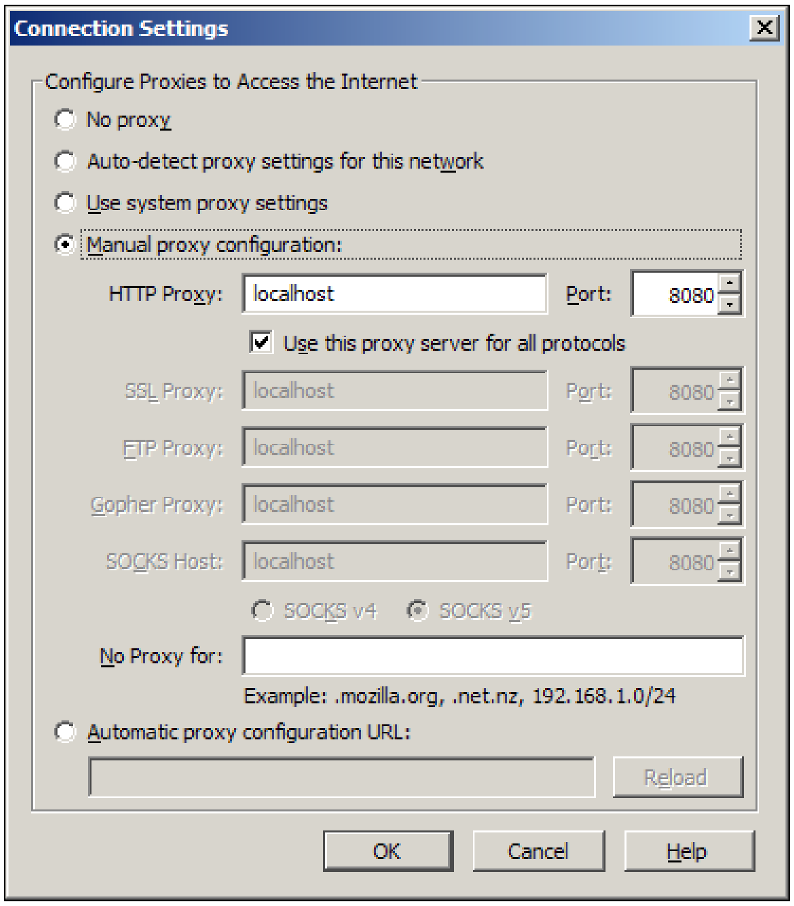
Port: 8080 Attempt HTTPS Spoofing Only spoof URLs matching:

Test plan content

Target Controller: Test Plan > Thread Group Grouping: Do not

[✓] Capture HTTP Headers [ ] Add Assertions [ ] Regex matching

When you are done, set up your web browser to use the JMeter proxy you just configured. In Firefox 3.6.x, this hides under **Tools** | **Options** | **Advanced** | **Network** | **Settings**. Choose `localhost` as the server and the port you defined in JMeter:



Connection Settings

Configure Proxies to Access the Internet

- No proxy
- Auto-detect proxy settings for this network
- Use system proxy settings
- [✓ Manual proxy configuration:]

HTTP Proxy: localhost Port: 8080

[✓] Use this proxy server for all protocols

SSL Proxy: localhost Port: 8080

FTP Proxy: localhost Port: 8080

Gopher Proxy: localhost Port: 8080

SOCKS Host: localhost Port: 8080

SOCKS v4 [✓ SOCKS v5]

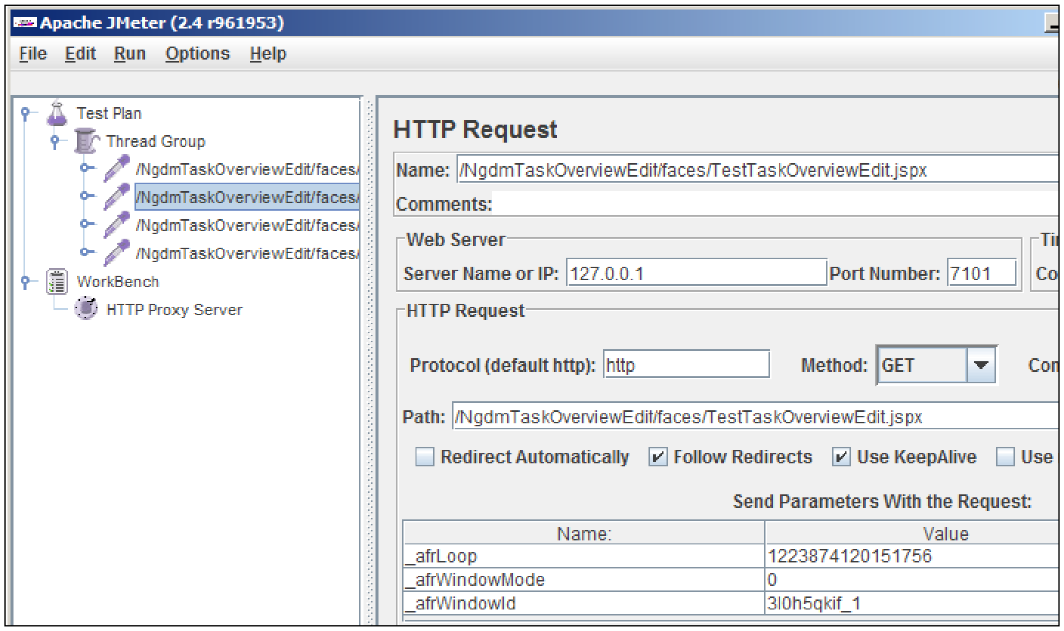
No Proxy for:

Example: .mozilla.org, .net.nz, 192.168.1.0/24

- Automatic proxy configuration URL:

Recording a session

To start recording, click **Start** at the bottom of the **HTTP Proxy Server** window in JMeter. Then go to your browser and enter the URL of the page you wish to test. You will see the page as normal in the browser. At the same time, everything is being recorded by JMeter—if you expand your thread group in JMeter, you will see that even a very simple page request might lead to multiple requests from the browser to the server:


Apache JMeter (2.4 r961953)
File Edit Run Options Help
Test Plan
Thread Group
/NgdmTaskOverviewEdit/faces/
/NgdmTaskOverviewEdit/faces/
/NgdmTaskOverviewEdit/faces/
/NgdmTaskOverviewEdit/faces/
WorkBench
HTTP Proxy Server
HTTP Request
Web Server
HTTP Request
Redirect Automatically
Follow Redirects
Use KeepAlive
Use
Send Parameters With the Request:

Name: /NgdmTaskOverviewEdit/faces/TestTaskOverviewEdit.jspx
Comments:
Server Name or IP: 127.0.0.1
Port Number: 7101
Protocol (default http): http
Method: GET
Path: /NgdmTaskOverviewEdit/faces/TestTaskOverviewEdit.jspx
☐ Redirect Automatically ☑ Follow Redirects ☑ Use KeepAlive ☐ Use

Name:	Value
_afrLoop	1223874120151756
_afrWindowMode	0
_afrWindowId	3l0h5qkif_1

When you have recorded what you want, you can stop the JMeter proxy again. Remember to also reconfigure your browser—you cannot access anything if the browser is pointing to a JMeter proxy that no longer runs.

Post-processing a recorded session

The recording shows the entire communication between the browser and the application server. If you examine the individual requests, you will see that after the initial request, the subsequent requests include one or more parameters—either with the request or as part of the URL.

In order to replay a session, you need to modify it so that JMeter can retrieve parameter values from one request and send them as part of the next request. The whole procedure involves:

- Adding a Cookie Manager
- Defining JMeter variables to hold the values from request to request
- Extracting values from requests into the variables
- Replacing the recorded values with references to the variables

Search for updated information

Using JMeter with ADF is based on reverse engineering the way ADF works—this is not officially documented or supported by Oracle. These instructions are based on work done by *Oracle ACE Director Chris Muir* and published on his blog at `http://one-size-doesnt-fit-all.blogspot.com`. They were valid at the time of writing, in JDeveloper 11.1.1.4, but since ADF internals might change without notice, you might need to search the Internet for updated instructions.

Adding a Cookie Manager

The Cookie Manager will receive cookies from the application server and provide the cookie values subsequent requests on demand. This is the easy part—just right-click on the **Thread Group** and choose **Add** | **Config Element** | **HTTP Cookie Manager**.

Defining variables

If you look at the URL of an ADF application, you will notice that it contains a number of name/value pairs after the question mark. These are used by ADF to ensure that each browser running the application will be connected to the right session on the server. In order for JMeter to be able to simulate a browser, it needs to retrieve some values from each server response in order to present the following request.

ADF uses the following variables that you need to define in JMeter by right-clicking on the **Thread Group** and choosing **Add** | **Config Element** | **User Defined Variables**:

- `afrLoop`
- `afrWindowId`
- `jsessionId`
- `adf.ctrl-state`
- `javax.faces.ViewState`

Extracting values

To get values from the requests and into your variables, you need to define five regular expression extractors by right-clicking on the **Thread Group** and choosing **Add** | **Post Processors** | **Regular Expression Extractor**:

- `afrLoop` extractor, retrieving `_afrLoop=([-_0-9A-Za-z]{13,16})` into the `afrLoop` variable
- `afrWindowId` extractor, retrieving `window.name='([-_0-9A-Za-z!]{10,13})'` into the `afrWindowId` variable
- `jsessionId` extractor, retrieving `;jsessionid=([-_0-9A-Za-z!]{63})` into the `jsessionId` variable
- `adf.ctrl-state` extractor, retrieving `_adf.ctrl-state=([-_0-9A-Za-z!]{10,13})` into the `adf.ctrl-state` variable
- `javax.faces.ViewState` extractor, retrieving `<input type="hidden" name="javax\.faces\.ViewState" value="!(.+?)">` into the `javax.faces.ViewState` variable

Fixing the path and the parameters

Now you have variables for all the values ADF passes back and forth—the final step is to go through every step in your JMeter test plan to check the following:

- The specific value in all references to `jsessionId` in the **Path** field must be changed to `${jsessionId}`.
- The specific value in all references to `_adf.ctrl-state` in the **Path** field must be changed to `${adf.ctrl-state}`.
- The specific value in all references to `_adf.ctrl-state`, `_afrLoop`, and `_afrWindowId` in the **Parameters** table must be changed to the corresponding variables. Note that `_afrWindowId` will have the literal value `null` in the first request—do not replace this value.
- The specific value in all references to `javax.faces.ViewState` in the **Parameters** table must be changed to `!${javax.faces.ViewState}`. Note the exclamation mark before this value. You also need to *deselect* the **Encode** checkbox for this parameter.
- The value for the parameter called `unique` must be replaced with the following expression: `${__javaScript(new Date().getTime(),DUMMY)}`.

Running a recorded session

To see what actually happens during the execution, you will add one or more Listeners. Right-click on the **Thread Group** and choose **Add | Listener** and choose one or more of the available listeners, for example:

- The **View Results in Table** listener shows an overview with response time and status
- The **View Results Tree** listener shows the detailed responses from the web server—normally, this should show **HTTP/1.1 200 OK** to indicate a successful HTTP request
- The **Graph Results** listener shows your response time graphically

It is possible to add JMeter assertions to verify that you are getting the right information back, but if you use Selenium or similar, this is not necessary.

When you have done this, you can run your recorded session by choosing **Run | Start** and watch the results roll in.

If you run an automated build and test process, JMeter testing is normally not included in the daily build, because application performance does not change from day-to-day. Instead, JMeter tests are normally run through the JMeter application once a large part of the application is complete. Preferably, your stress/performance tests should be executed on a dedicated test environment similar to the production environment.

The Oracle alternative

If you would rather purchase an integrated testing solution than putting together these open source tools, Oracle offers Oracle Application Test Suite.

The product consists of:

- *Oracle Functional Testing* for testing your application functionality (like Selenium does)
- *Oracle Load Testing* for testing how your application performs under load (like JMeter does)
- *Oracle Test Manager* for managing the test process (there is no equivalent open source tool)

You can find more information about Oracle Application Test Suite at `http://www.oracle.com/technetwork/oem/app-test/index.html`.

Summary

Together with the test team in DMC solutions, you have created unit tests for your business components and user interface tests that exercise the whole XDM application. Once the application passed these tests, you recorded a stress test and ran it to be sure that the application performs as you expect it to, also under heavy load.

Functionally, the XDM application is ready to deploy but your boss would like you to tweak the user interface just a little before releasing the application. That is the topic of the next chapter.

8
Look and Feel

Henry Ford famously said: *Any customer can have a car painted any color that he wants so long as it is black*. He actually had a good reason: Black paint dried quicker, allowing him to operate the assembly line at a higher speed.

Today, you can have your car in a large number of standard colors. And, you can have your web browser in any color you want—Firefox now offers more than 35,000 *personas* with different colors and images.

You probably do not want to offer your enterprise application in thousands of color schemes, but you do want to build a visually attractive application that matches the graphical identity used by your organization. And if your application is going to be used by several different groups of users, for example, if it is going to be deployed to multiple customers—you want it to be easy to change the look and feel of the application.

Controlling the appearance

An *ADF Faces* application is a modern web application, so the technology used for controlling the look of the application is **Cascading Style Sheets** (**CSS**).

The idea behind CSS is that the web page (in HTML) should contain only structure and not information about the appearance. All of the visual definitions must be kept in the style sheet, and the HTML file must refer to the style sheet. It follows that the same web page can be made to look completely different by applying a different style sheet to it.

Cascading Style Sheets basics

In order to be able to change the appearance of your application, you need to understand some CSS basics. If you have never worked with CSS before, you should start by reading one of the many CSS tutorials available on the Internet.

To establish the basis for the following discussion, let us repeat some of the basics of CSS.

The CSS layout instructions are written in the form of **rules**. Each rule is of the form:

```
selector { property: value; }
```

The **selector** identifies which part of the web page the rule applies to, and the **property/value** pairs define the styling to be applied to the selected parts.

For example, the rule:

```
h1 { color: red; }
```

defines that all `<h1>` elements should be shown in red font.

One rule can include multiple selectors, separated by commas, and multiple property values, separated by semicolons. Therefore, it is also valid CSS to write:

```
h1, h2, h3 { color: red; font-size: x-large; }
```

to get all the `<h1>`, `<h2>`, and `<h3>` tags shown in large, red font.

If you want to apply a style with more precision than just every level 1 header, you define a **style class**, which is just a selector starting with a period:

```
.important { color: red; font-weight: bold }
```

To use this selector in your HTML code, you use the keyword `class` inside an HTML tag. There are three ways of using a style class:

- Inside an existing tag such as `<h1>`
- Inside the special `<span>` tag for styling text within a paragraph
- Inside a `<div>` tag to style a whole paragraph of text

Here are the examples of all three ways:

```
<h1 class="important">Important topic</h1>
You <span class="important">must</span> remember this.
<div class="important">Important tip</div>
```

In theory, you can place your styling information directly in your HTML document using the `<style>` tag. In practice, however, you usually place your CSS instructions in a separate `.css` file and refer to it from your HTML file with a `<link>` tag, as follows:

```
<link href="mystyle.css" rel="stylesheet" type="text/css">
```

Styling individual components

The preceding examples are applied to the HTML elements, but styling can also be applied to JSF components. A plain JSF component could look like the following with inline styling:

```
<h:outputFormat value="hello" style="color:red;"/>
```

Or like the following using a style class:

```
<h:outputFormat value="hello" styleClass="important"/>
```

The ADF components use the attribute `inlineStyle` instead of just `style`:

```
<af:outputFormat value="hello" inlineStyle="color:red;"/>
```

The `styleClass` attribute is the same:

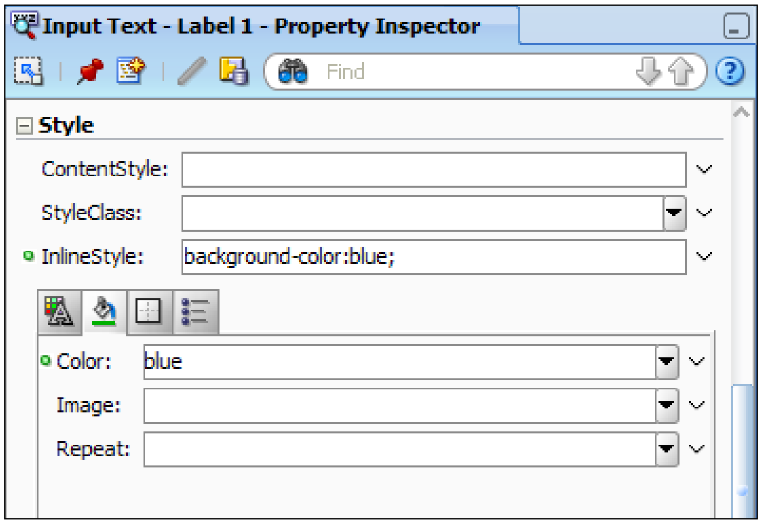
```
<af:outputFormat value="hello" styleClass="important"/>
```

Of course, you normally will not be setting these attributes in the source code, but instead will be using the **StyleClass** and **InlineStyle** properties in the **Property Inspector**.

In both HTML and JSF, you should only use **StyleClass** so that multiple components can refer to the same style class and will all reflect any change made to the style. **InlineStyle** is rarely used in real-life ADF applications; it adds to the page size (same styling is sent for every styled element) and it is almost impossible to ensure that every occurrence is changed when the styling requirements change—as they will.

Building a Style

While you are working out the styles you need in your application, you can use the **Style** section in the JDeveloper **Property Inspector** to define the look of your page. This section shows four small sub-tabs with icons for **Text**, **Background**, **Box**, and **Classification**. If you enter or select a value in any of these fields, this value will be placed into the **InlineStyle** field as correctly formatted CSS:



ContentStyle:
StyleClass:
InlineStyle: background-color:blue;
Color: blue
Image:
Repeat:

When your items look the way you want, copy the value from the **InlineStyle** field to a style class in your CSS file and set the **StyleClass** property to point to that class. If the preceding is the styling you want for a highlighted label, create a section in your CSS file as follows:

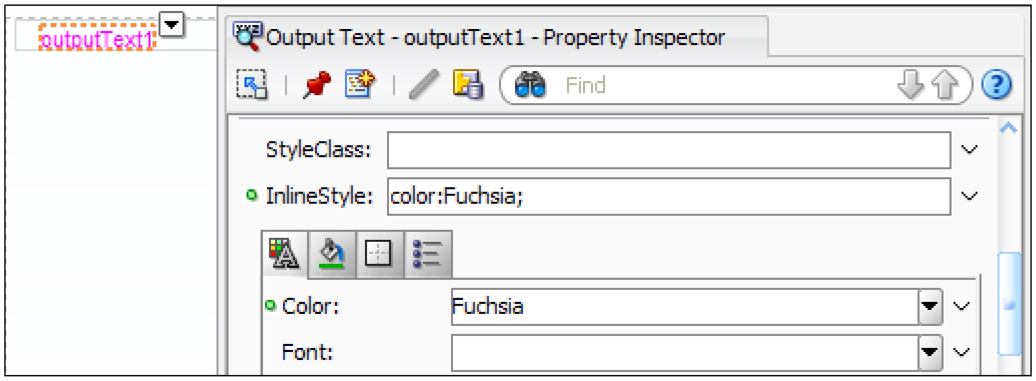
```
.highlight {background-color:blue;}
```

Then clear the **InlineStyle** property and set the **StyleClass** property to **highlight**. From now on, you can style other components in exactly the same way by just setting the **StyleClass** property.

We will be building the actual CSS file where you define these style classes in the section on skinning later in this chapter.

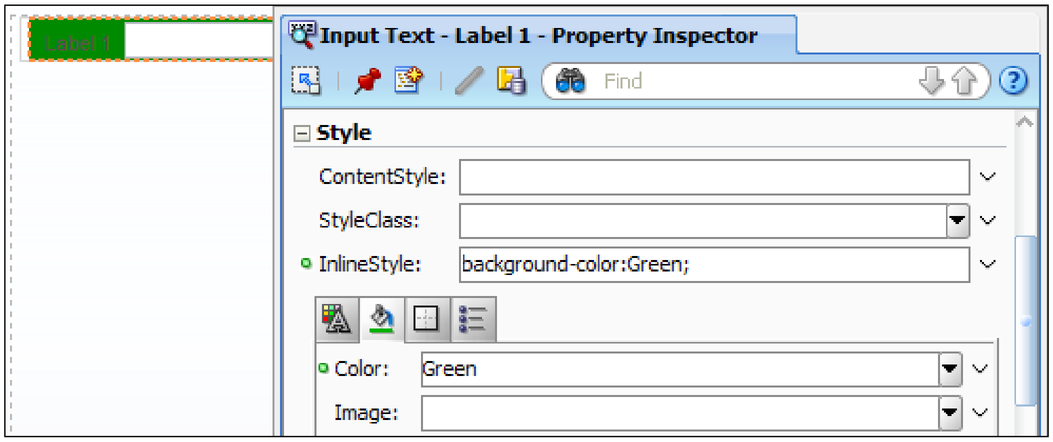
InlineStyle and ContentStyle

Some JSF components (for example, `outputText`) are easy to style. If you set the font color, you will see it take effect in the JDeveloper design view and in your application:



StyleClass:
InlineStyle: color:Fuchsia;
Color: Fuchsia
Font:

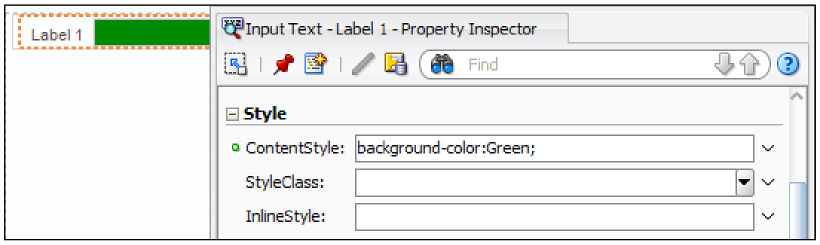
Other elements (for example, `inputText`) are harder to style. For example, if you want to change the background color of the input field, you might try setting the background color:



ContentStyle:
StyleClass:
InlineStyle: background-color:Green;
Color: Green
Image:

You see that this did *not* work the way you probably expected—there is a green background behind both the label and the actual input field. The reason for this is that an **inputText** component actually consists of several HTML elements, and an inline style applies to the outermost element. In this case, the outermost element is an HTML `<tr>` (table row) tag, so the green background color applies to the entire row.

To help mitigate this problem, ADF offers another styling option for some components: **ContentStyle**. If you set this property, ADF tries to apply the style to the content of a component—in the case of an **inputText**, **ContentStyle** applies to the actual input field:



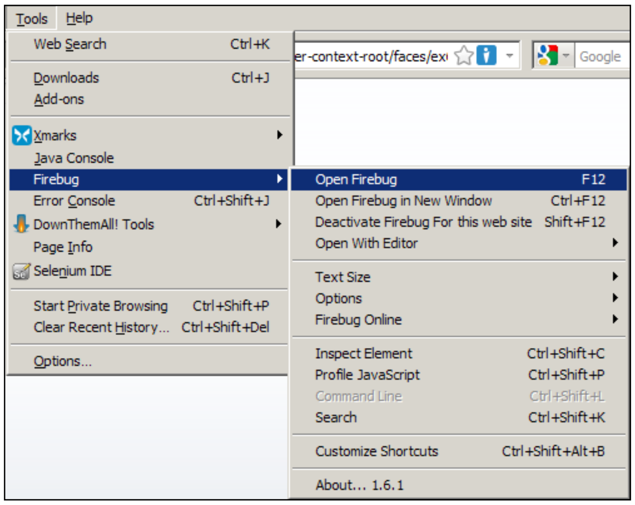
ContentStyle: background-color:Green;
StyleClass:
InlineStyle:

This will help in some cases, but for instance, if you want to apply styling to the label for an **inputText**, you will have to use a *skin* as discussed later in this chapter.

Why does it look like that?

As you saw in the preceding **inputText** example, ADF components can be quite complex and it is not always easy to figure out which element to style to achieve the desired result. To be able to see into the complex HTML that ADF builds for you, you need a support tool such as **Firebug**. Firebug is a Firefox extension that you can download with **Tools | Add-ons** from within Firefox, or you can go to `http://getfirebug.com`.

When you have installed Firebug, you see a new **Firebug** sub-menu on your **Tools** menu in Firefox:

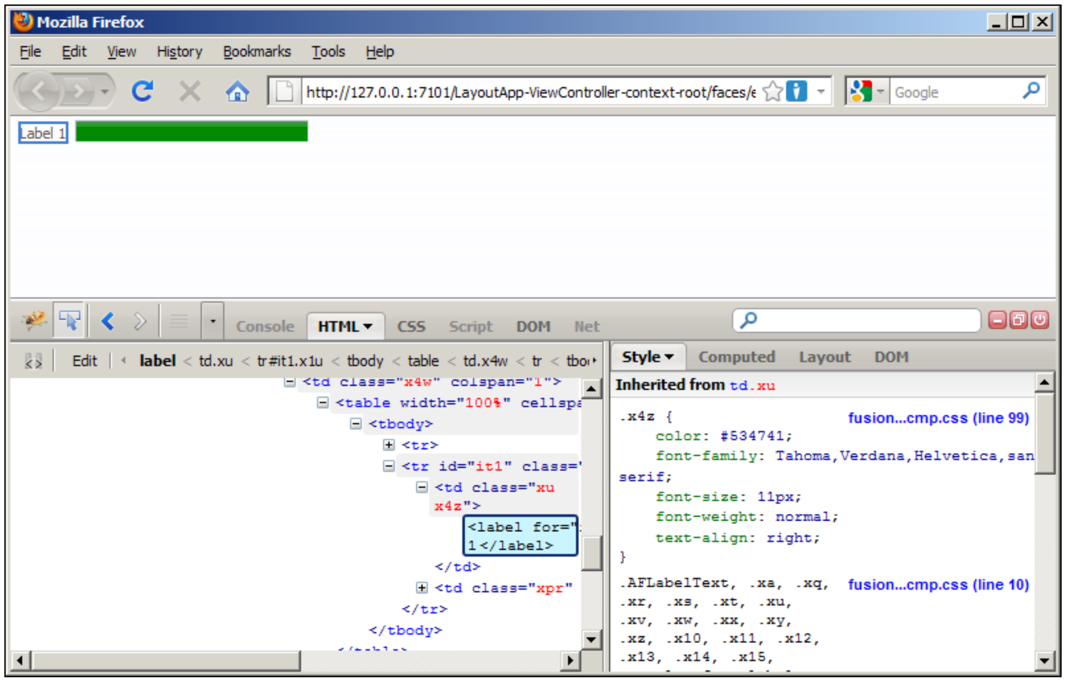

Tools
Help
Web Search Ctrl+K
Downloads Ctrl+J
Add-ons
Xmarks
Java Console
Firebug
Error Console Ctrl+Shift+J
DownThemAll! Tools
Page Info
Selenium IDE
Start Private Browsing Ctrl+Shift+P
Clear Recent History... Ctrl+Shift+Del
Options...
er-context-root/faces/ex
Google
Open Firebug F12
Open Firebug in New Window Ctrl+F12
Deactivate Firebug For this web site Shift+F12
Open With Editor
Text Size
Options
Firebug Online
Inspect Element Ctrl+Shift+C
Profile JavaScript Ctrl+Shift+P
Command Line Ctrl+Shift+L
Search Ctrl+Shift+K
Customize Shortcuts Ctrl+Shift+Alt+B
About... 1.6.1

When you start Firebug, you will see it take up the lower half of your Firefox browser window.

Only run Firebug when you need it

It probably does not surprise you that Firebug's detailed analysis of every page costs processing power and slows your browser down. Only run Firebug when you need it.

If you click on the **Inspect** button (with a little blue arrow, second from left in the Firebug toolbar), you place Firebug in *inspect* mode. You can now point to any element on a page and see both the HTML element and the style applied to this element:

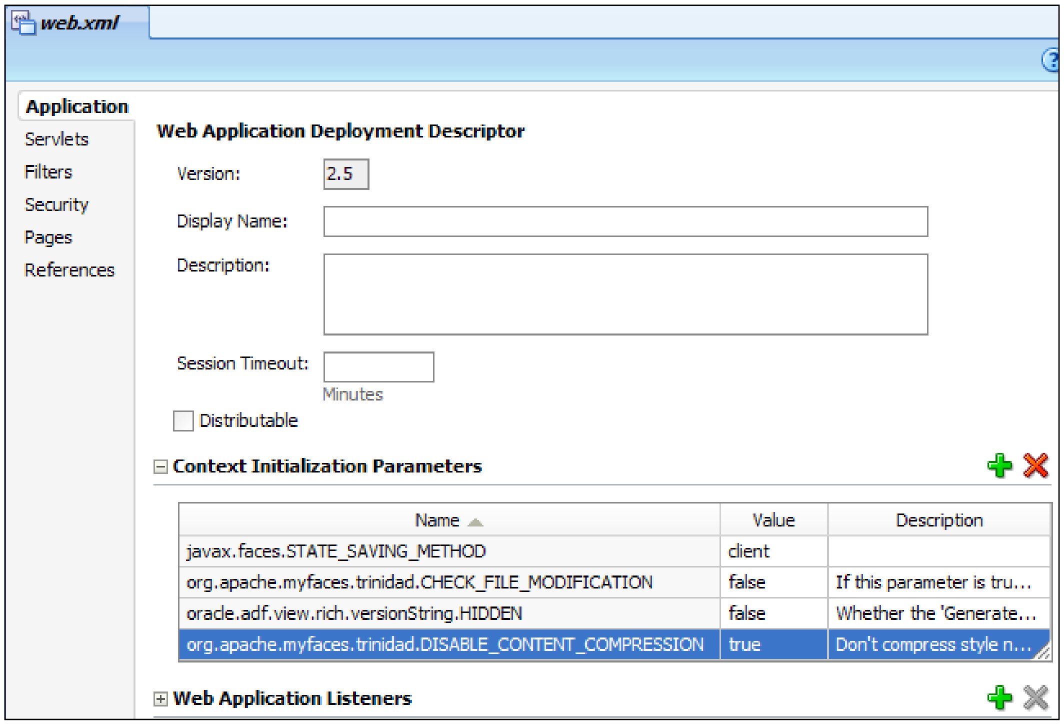

Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://127.0.0.1:7101/LayoutApp-ViewController-context-root/faces/e
Google
Label 1
Console HTML CSS Script DOM Net
Edit label < td.xu < tr#it1.x1u < tbody < table < td.x4w < tr < tbo
Style Computed Layout DOM

```
<td class="x4w" colspan="1">
  <table width="100%" cellspa
    <tbody>
      <tr>
      <tr id="it1" class='
        <td class="xu
        x4z">
          <label for="
          1</label>
        </td>
        <td class="xpr"
      </tr>
    </tbody>
```

```
Inherited from td.xu
.x4z {                          fusion...cmp.css (line 99)
    color: #534741;
    font-family: Tahoma,Verdana,Helvetica,san
serif;
    font-size: 11px;
    font-weight: normal;
    text-align: right;
}
.AFLabelText, .xa, .xq,         fusion...cmp.css (line 10)
.xr, .xs, .xt, .xu,
.xv, .xw, .xx, .xy,
.xz, .x10, .x11, .x12,
.x13, .x14, .x15,
```

In the preceding example, the pointer is placed on the label for an **inputText**, and the Firebug panels show that this element is styled with **color: #534741**, has **font-size: 11px**, and so on.

In order to keep the size of the HTML page down so that it loads faster, ADF has abbreviated all the style class names to cryptic short names such as `.x4z`. While you are styling your application, you do not want this abbreviation to happen. To turn it off, you need to open the `web.xml` file (in your **View** project under **Web Content**, **WEB-INF**), change to the **Overview** tab if it is not already shown, and select the **Application** sub-tab:



Version: 2.5
Display Name:
Description:
Session Timeout:

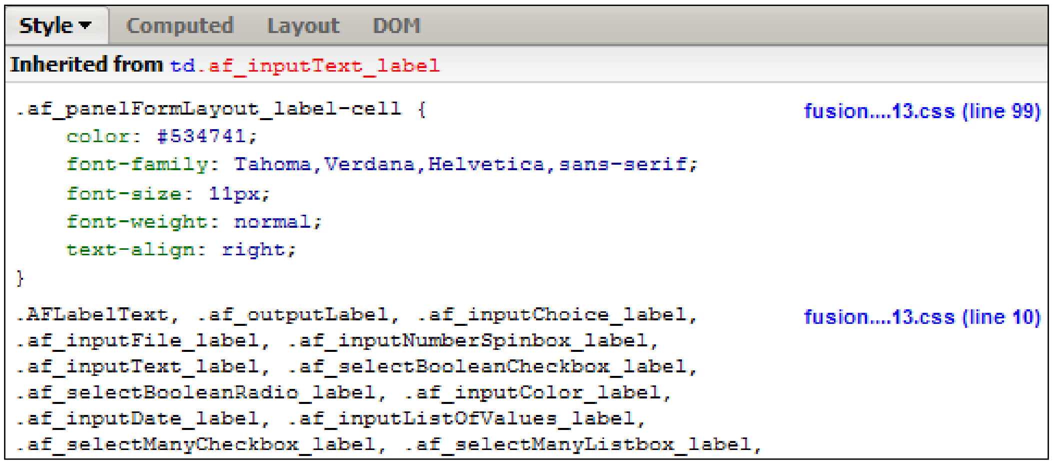
Context Initialization Parameters

Name	Value	Description
javax.faces.STATE_SAVING_METHOD	client	
org.apache.myfaces.trinidad.CHECK_FILE_MODIFICATION	false	If this parameter is tru...
oracle.adf.view.rich.versionString.HIDDEN	false	Whether the 'Generate...
org.apache.myfaces.trinidad.DISABLE_CONTENT_COMPRESSION	true	Don't compress style n...

Under **Context Initialization Parameters**, add a new parameter:

- **Name**: `org.apache.myfaces.trinidad.DISABLE_CONTENT_COMPRESSION`
- **Value**: `true`

When you do this, you will see the full human-readable style names in Firebug:



Inherited from td.af_inputText_label

```
.af_panelFormLayout_label-cell {                    fusion....13.css (line 99)
    color: #534741;
    font-family: Tahoma,Verdana,Helvetica,sans-serif;
    font-size: 11px;
    font-weight: normal;
    text-align: right;
}
.AFLabelText, .af_outputLabel, .af_inputChoice_label,   fusion....13.css (line 10)
.af_inputFile_label, .af_inputNumberSpinbox_label,
.af_inputText_label, .af_selectBooleanCheckbox_label,
.af_selectBooleanRadio_label, .af_inputColor_label,
.af_inputDate_label, .af_inputListOfValues_label,
.af_selectManyCheckbox_label, .af_selectManyListbox_label,
```

You will notice that the cryptic `.x4z` style class is really the **.af_panelFormLayout_label-cell** style class. You might need this information when developing your custom skin.

Getting back up to speed

If you want this compression enabled in your production application for best performance, remember to remove this initialization parameter again before deploying your application to production.

Conditional formatting

Like many other properties, the style properties do not have to be set to a fixed value—you can also set them to any valid language expression. This can be used to create conditional formatting.

In the simplest form, you can use an expression language **ternary operator**, which has the form `<boolean expression> ? <value if true > : <value if false>`. For example, you could set **StyleClass** to:

```
#{bindings.Job.inputValue eq 'MANAGER' ? 'managerStyle' :
'nonManagerStyle'}
```

This expression means that if the value of the `Job` attribute is equal to `MANAGER`, use the **managerStyle** style class, if not, use the **nonManagerStyle** style class. Of course, this only works if these two styles exist in your CSS style sheet.

Skinning

An **ADF skin** is a collection of files that together define the look and feel of the application. To a hunter, skinning is the process of removing the skin from an animal, but to an ADF developer, it is the process of putting a skin onto an application.

All applications have a skin—if you do not change it, an application built with JDeveloper 11g uses some variation of the *fusion* skin, which is also used for Oracle Fusion Applications.

When you define a custom skin, you must also choose a parent skin among the skins JDeveloper offers. This parent skin will define the look for all components not explicitly defined in your skin.

What should I skin?

If your graphics designer has produced sample screens showing what the application must look like, you need to find out which components you will use to implement the required look, and define the look of these components in your skin.

If you do not have a detailed guideline from a graphics designer, look for some guidelines in your organization; you probably have web design guidelines for your public-facing website and/or intranet.

If you do not have any graphics guidelines, create a skin as described later in this section and choose to inherit from the latest *fusion* skin provided by Oracle. But do not change anything—leave the CSS file empty. If you are a programmer like me, you are unlikely to be able to improve on the look that the professional graphics designers in Redwood Shores have created.

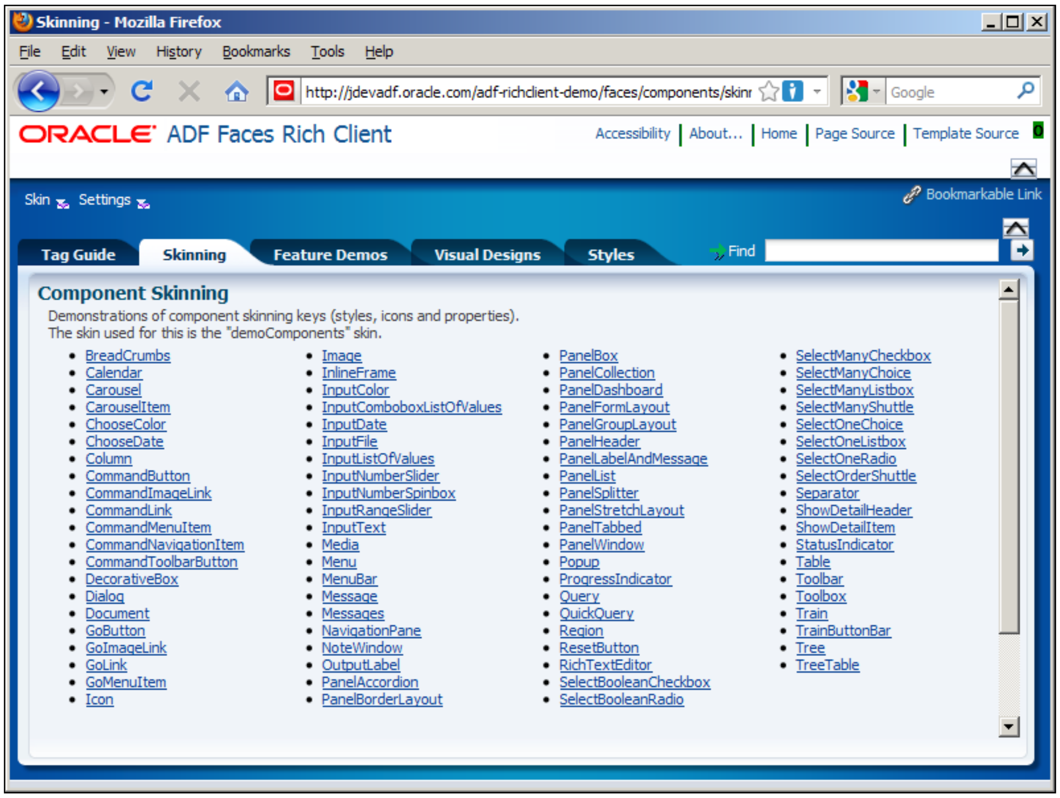
Always wear your own skin

Creating an empty skin is similar to creating an empty framework extension classes as you did in *Chapter 5, Prepare to Build*. It provides a placeholder that you can fill with content later if you need it.

What can I skin?

The section on styling components mentioned that some aspects of the application could not be changed using the style properties, for example, the font used for labels inside a **PanelFormLayout**. With a custom skin, however, you can change every aspect of your application.

To see skinning in action, you can go to `http://jdevadf.oracle.com/adf-richclient-demo`. This site is a demonstration of lots of ADF features and components, and if you choose the **Skinning** tab, you are presented with a long list of skinnable components:


Skinning - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://jdevadf.oracle.com/adf-richclient-demo/faces/components/skinr
Google
ORACLE ADF Faces Rich Client
Accessibility | About... | Home | Page Source | Template Source
Skin Settings
Bookmarkable Link
Tag Guide
Skinning
Feature Demos
Visual Designs
Styles
Find

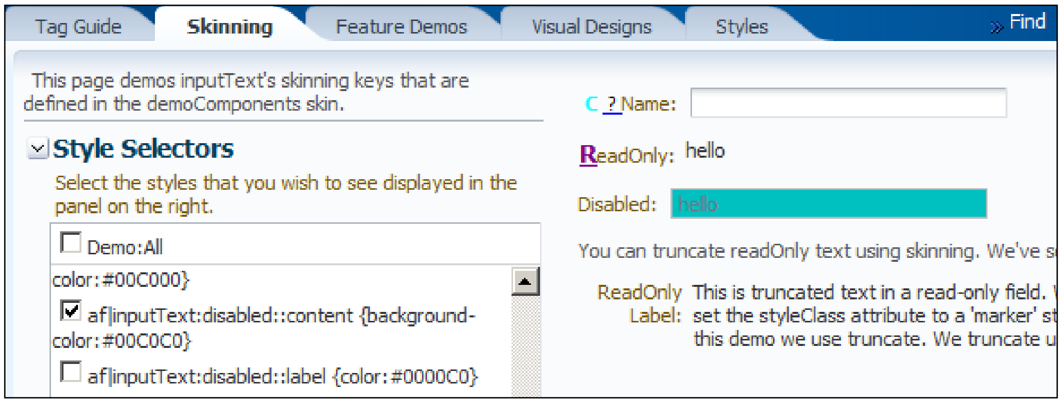
Component Skinning

Demonstrations of component skinning keys (styles, icons and properties).
The skin used for this is the "demoComponents" skin.

- BreadCrumbs
- Calendar
- Carousel
- CarouselItem
- ChooseColor
- ChooseDate
- Column
- CommandButton
- CommandImageLink
- CommandLink
- CommandMenuItem
- CommandNavigationItem
- CommandToolbarButton
- DecorativeBox
- Dialog
- Document
- GoButton
- GoImageLink
- GoLink
- GoMenuItem
- Icon
- Image
- InlineFrame
- InputColor
- InputComboboxListOfValues
- InputDate
- InputFile
- InputListOfValues
- InputNumberSlider
- InputNumberSpinbox
- InputRangeSlider
- InputText
- Media
- Menu
- MenuBar
- Message
- Messages
- NavigationPane
- NoteWindow
- OutputLabel
- PanelAccordion
- PanelBorderLayout
- PanelBox
- PanelCollection
- PanelDashboard
- PanelFormLayout
- PanelGroupLayout
- PanelHeader
- PanelLabelAndMessage
- PanelList
- PanelSplitter
- PanelStretchLayout
- PanelTabbed
- PanelWindow
- Popup
- ProgressIndicator
- Query
- QuickQuery
- Region
- ResetButton
- RichTextEditor
- SelectBooleanCheckbox
- SelectBooleanRadio
- SelectManyCheckbox
- SelectManyChoice
- SelectManyListbox
- SelectManyShuttle
- SelectOneChoice
- SelectOneListbox
- SelectOneRadio
- SelectOrderShuttle
- Separator
- ShowDetailHeader
- ShowDetailItem
- StatusIndicator
- Table
- Toolbar
- Toolbox
- Train
- TrainButtonBar
- Tree
- TreeTable

You can click on each component to see a page where you can experiment with various ways of skinning the component.

For example, you can select the very common **inputText** component to see a page with various representations of the **inputText** components. On the left, you see a number of **selectors** that are relevant for that component. For each selector, you can check the checkbox to see an example of what the component looks like if you change that selector. In the following example, the **af|inputText:disabled::content** selector is checked, setting the style for this selector to **color: #0000C0**:



This page demos inputText's skinning keys that are defined in the demoComponents skin.

Style Selectors

Select the styles that you wish to see displayed in the panel on the right.

☐ Demo:All

color:#00C000}

☑ af|inputText:disabled::content {background-color:#00C0C0}

☐ af|inputText:disabled::label {color:#0000C0}

C ? Name:

ReadOnly: hello

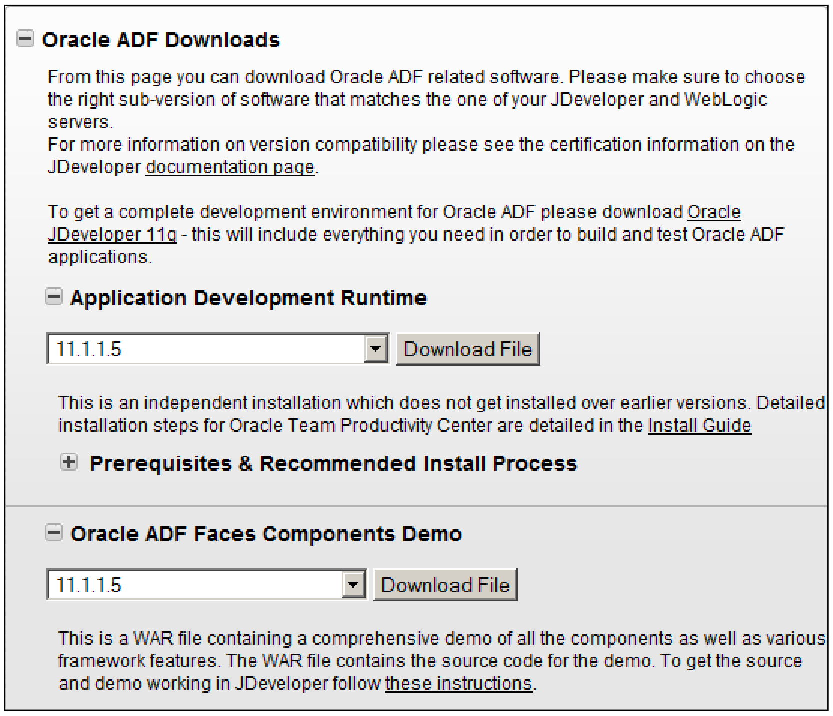
Disabled: hello

You can truncate readOnly text using skinning. We've s

ReadOnly Label: This is truncated text in a read-only field. set the styleClass attribute to a 'marker' st this demo we use truncate. We truncate u

As you might be able to deduce from the **af|inputText:disabled::content** style selector, this controls what the *content* field of the *input text* component looks like when it is set to *disabled*—in the demo application, it is set to a bluish green color with the color code **#0000C0**. The example application shows various values for the selectors, but does not really explain them. The full documentation of all selectors can be found online—at the time of writing at `http://jdevadf.oracle.com/adf-richclient-demo/docs/skin-selectors.html`. If it is not there, search for **ADF skinning selectors**.

In the top-left corner, you will also find a **Skin** drop-down that you can use to select and test all the built-in skins. This application can also be downloaded and run on your own server—at the time of writing, it could be found on the ADF download page at `http://www.oracle.com/technetwork/developer-tools/adf/downloads/index.html`:


Oracle ADF Downloads

From this page you can download Oracle ADF related software. Please make sure to choose the right sub-version of software that matches the one of your JDeveloper and WebLogic servers.
For more information on version compatibility please see the certification information on the JDeveloper documentation page.

To get a complete development environment for Oracle ADF please download Oracle JDeveloper 11g - this will include everything you need in order to build and test Oracle ADF applications.

Application Development Runtime

11.1.1.5 Download File

This is an independent installation which does not get installed over earlier versions. Detailed installation steps for Oracle Team Productivity Center are detailed in the Install Guide

Prerequisites & Recommended Install Process

Oracle ADF Faces Components Demo

11.1.1.5 Download File

This is a WAR file containing a comprehensive demo of all the components as well as various framework features. The WAR file contains the source code for the demo. To get the source and demo working in JDeveloper follow these instructions.

Skinning overview

The skinning process in ADF consists of the following steps:

1. Creating a skin CSS file.
2. Optionally providing images for your skin.
3. Optionally changing the color scheme for your skin.
4. Optionally creating a resource bundle for your skin.
5. Packaging the skin in an ADF Library.
6. Importing and using the skin in the application.

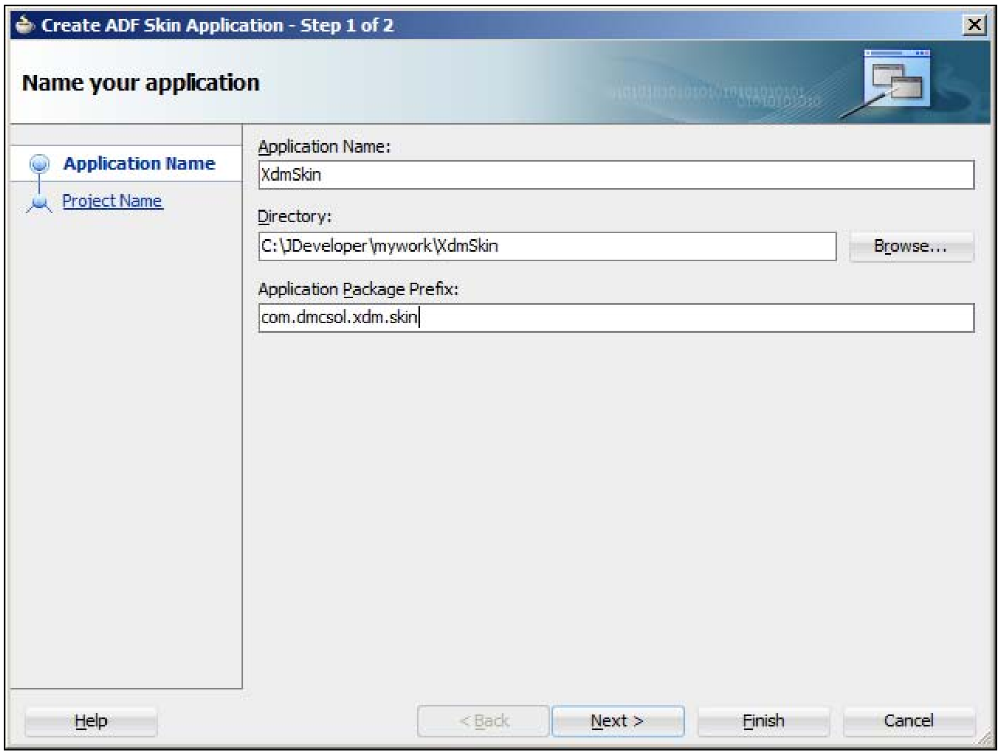
Up to and including the 11g Release 1 versions of JDeveloper (with a 11.1.1.x version number), this was very much a manual process. Fortunately, JDeveloper 11g Release 2 has a skin editor built in.

Stand-alone skinning

If you are running JDeveloper 11g Release 1, do not despair. Oracle is making a stand-alone skin editor available, containing the same functionality as JDeveloper 11g Release 2. You can give this tool to your graphics designers and let them build the skin ADF Library without having to give them the full JDeveloper product. The screens shown in this chapter are captured from a pre-release version of the stand-alone skin editor.

Starting a skin

This description assumes the stand-alone skin editor. If you are running JDeveloper 11g Release 2, you can simply open the Common UI project inside your Common UI workspace and skip to the start of the next section. In the stand-alone skin editor, you start by creating a new application workspace by selecting **File | New | ADF Skin Application**. Give your application a name, choose a directory and provide an application package prefix:



Application Name: XdmSkin

Directory: C:\JDeveloper\mywork\XdmSkin

Application Package Prefix: com.dmcsol.xdm.skin

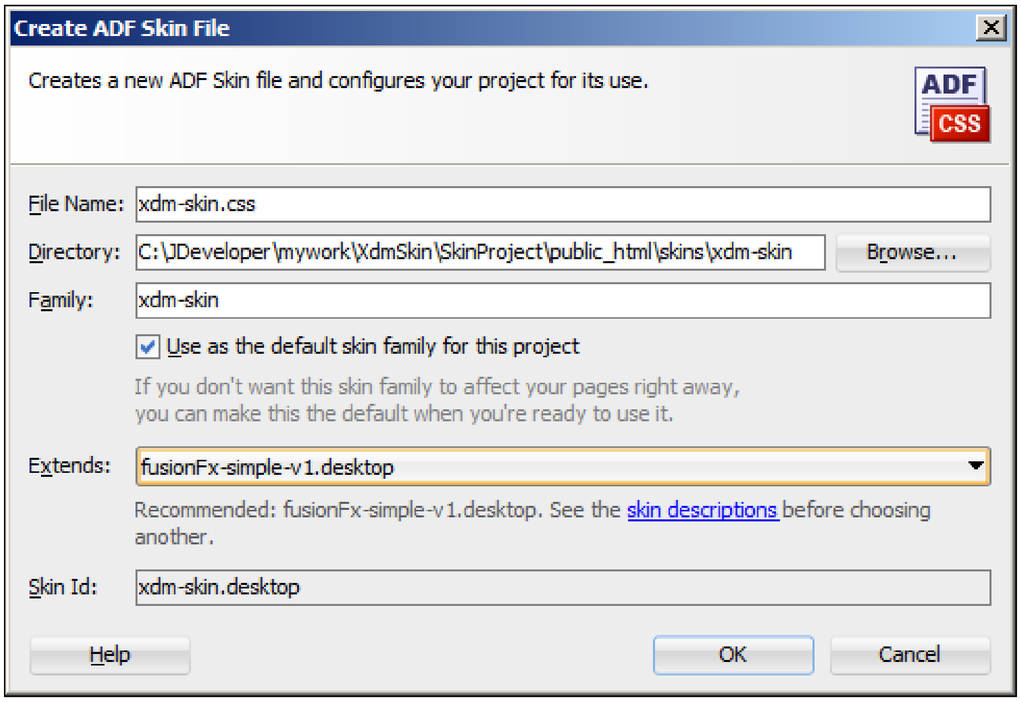
In step 2 of the **Create ADF Skin Application** wizard, you can just click **Finish**.

Creating a skin CSS file

The most important part of your skin is the special ADF CSS file that defines the look of all the components you use in your application.

Creating the CSS file

To create this CSS file, choose **File | New | ADF Skin File**. In the **Create ADF Skin File** dialog, give your CSS file a filename that includes `-skin` (for example `xdm-skin.css`). You can leave the location at the default, and you do not have to change the value for **Family** either:


Create ADF Skin File

Creates a new ADF Skin file and configures your project for its use.

File Name: xdm-skin.css

Directory: C:\JDeveloper\mywork\XdmSkin\SkinProject\public_html\skins\xdm-skin

Family: xdm-skin

[✓] Use as the default skin family for this project

If you don't want this skin family to affect your pages right away, you can make this the default when you're ready to use it.

Extends: fusionFx-simple-v1.desktop

Recommended: fusionFx-simple-v1.desktop. See the skin descriptions before choosing another.

Skin Id: xdm-skin.desktop

Skin families

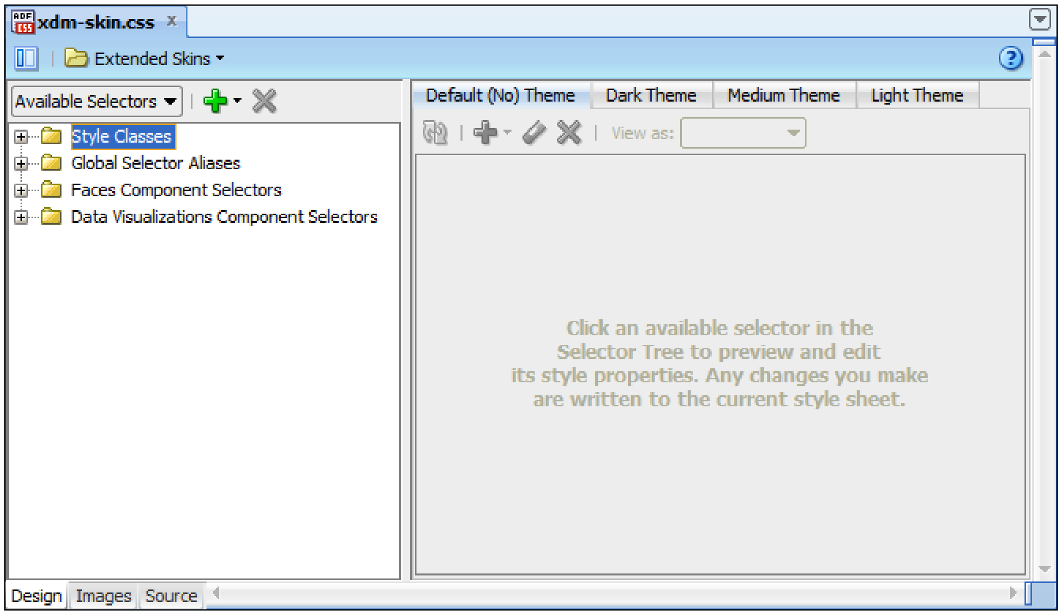
Skins come in **families**—a collection of related skins. All skins in a family share the same family name, and each skin in the family has a unique **Skin Id** consisting of the family name and a platform suffix. Start with the `.desktop` member of the family—this is the version of the skin that will be used for normal browser access to your application. You can also define other family members such as `.mobile` for controlling how the application will look on a mobile device.

Choose to extend **fusionFx-simple-v1.desktop** (the default). If you have another version than **v1** available, simply choose the latest and click **OK**.

Keeping it simple

The `fusionFx-v1` skin family has been developed explicitly for Oracle Fusion Applications. It contains a lot of features, but Oracle has realized that most enterprise applications are significantly simpler than Oracle Fusion Applications. That is why Oracle recommends you should inherit from the `fusionFx-simple-v1` skin, which has been optimized for normal enterprise applications and does not contain everything and the kitchen sink.

JDeveloper or the standalone skin editor automatically recognizes your ADF skin CSS file as a special kind of CSS and shows it in a special skin editor tab:



Click an available selector in the Selector Tree to preview and edit its style properties. Any changes you make are written to the current style sheet.

This dialog has four top-level nodes in the tree to the left:

- **Style Classes**
- **Global Selector Aliases**
- **Faces Component Selectors**
- **Data Visualizations Component Selectors**

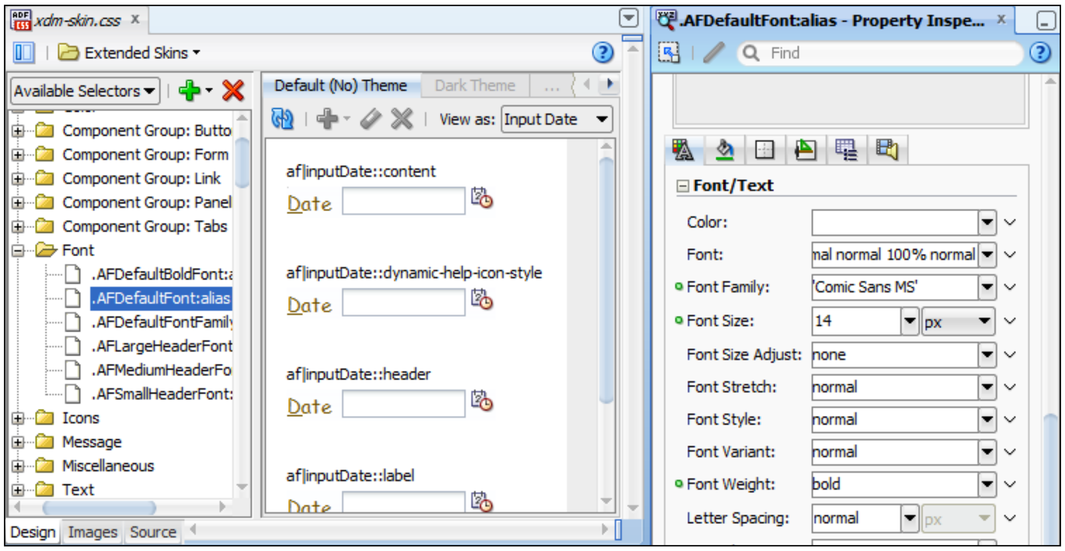
Style Classes

The **Style Classes** are built-in styles intended for your use in the **StyleClass** attribute—these do no affect any components.

If you need to define your own styles classes, you can click on the green plus sign and choose **New Style Class** to define a new class. It will be added to your CSS file and will show up under the **Style Classes** node. You can define the visual properties of the class using the **Property Inspector** or directly in the CSS source code by selecting the **Source** tab at the bottom of the skin editor panel.

Global Selector Aliases

The **Global Selector Aliases** are selectors that control many different components. For example, if you wanted to change the font throughout your entire application, you would open the **Font** node and select the **.AFDefaultFont:alias** selector. The right-hand side of the ADF CSS tab shows an example of how this global selector would affect various components. You can choose which components you want to examine from the **View as** drop-down list. In the **Property Inspector**, you can set the styling attributes for the global selector:



Component Group: Butto
Component Group: Form
Component Group: Link
Component Group: Panel
Component Group: Tabs
Font
.AFDefaultBoldFont:a
.AFDefaultFont:alias
.AFDefaultFontFamily
.AFLargeHeaderFont
.AFMediumHeaderFo
.AFSmallHeaderFont:
Icons
Message
Miscellaneous
Text

af|inputDate::content
Date
af|inputDate::dynamic-help-icon-style
Date
af|inputDate::header
Date
af|inputDate::label
Date

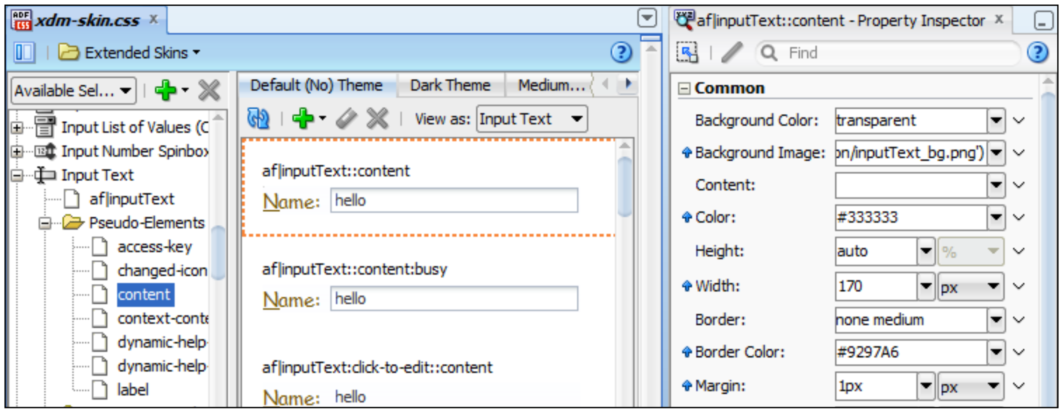
Color:
Font: nal normal 100% normal
Font Family: 'Comic Sans MS'
Font Size: 14 px
Font Size Adjust: none
Font Stretch: normal
Font Style: normal
Font Variant: normal
Font Weight: bold
Letter Spacing: normal px

In the preceding example, we have selected **.AFDefaultFont:alias** and set **Font Family** to '**Comic MS Sans**', **Font Size** to **14**, and **Font Weight** to **bold** (the changed attributes are marked with a green dot). The box in the middle shows how an **Input Date** would look with this setting. You can choose another component in the **View as** drop-down to see how this global setting would affect other types of components.

Faces Component Selectors

Under the **Faces Component Selectors** node, you find all the individual ADF Faces components that you can change the visual appearance of. To change a component, expand the node corresponding to the component to see the different selectors you can control.

You can click on the component itself to set the general attributes for the component, or you can expand the **Pseudo-Elements** node to change the specific aspects. For example, if you want to change the actual field where the user enters data, you can select the **content** pseudo-element. In the right-hand side of the ADF Skin dialog, you see the various sub-types of content styling that can apply to the component. As you select these sub-types, the **Property Inspector** shows the styling of that sub-type:



af|inputText::content
Name: hello

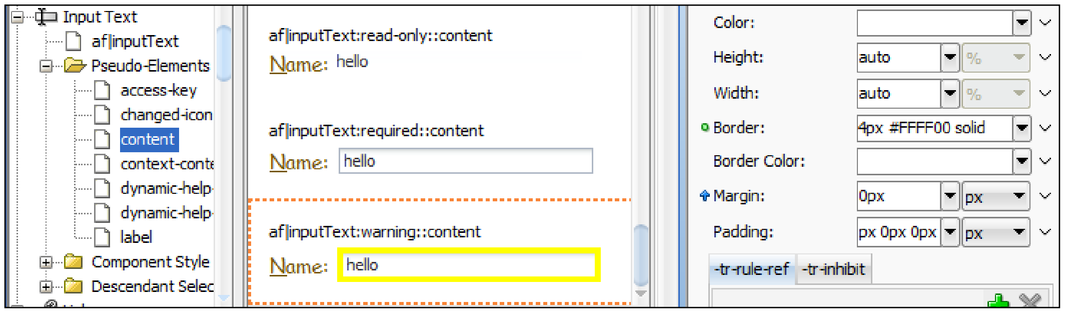
af|inputText::content:busy
Name: hello

af|inputText:click-to-edit::content
Name: hello

Background Color:	transparent	
Background Image:	on/inputText_bg.png')	
Content:		
Color:	#333333	
Height:	auto	%
Width:	170	px
Border:	none medium	
Border Color:	#9297A6	
Margin:	1px	px

The preceding example shows the default styling of the entry field for an input text component. The label is shown in Comic Sans MS font, because we set that as a global setting previously. Notice the blue arrows that indicate a setting that is inherited from somewhere else—you can point to the arrow to see a popup showing you where that setting inherits from.

For example, if you want to change the border for a text field that contains a warning, scroll down among the examples of **inputText** content to find the **af|inputText:warning::content** example and select it. In the **Property Inspector**, change the **Border** property, for example to **4px #FFFF00 solid** to use a wider border in bright yellow color:



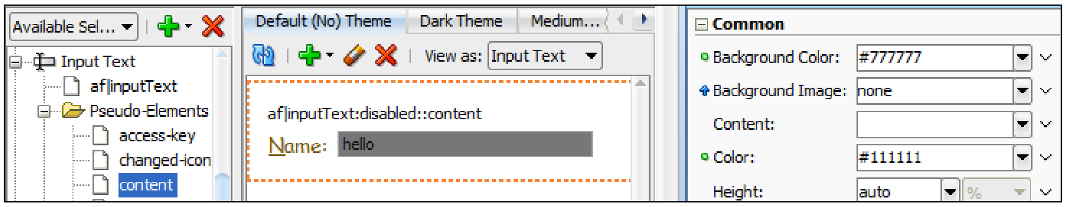
af|inputText:read-only::content
Name: hello

af|inputText:required::content
Name: hello

af|inputText:warning::content
Name: hello

Color:
Height: auto %
Width: auto %
Border: 4px #FFFF00 solid
Border Color:
Margin: 0px px
Padding: px 0px 0px px

Similarly, if you want to change a disabled input text field to display as dark gray with black text, you'd select the **af|inputText:disabled::content** example, change the **Background Color** property to **#777777** and **Color** (which indicates the font color) to **#111111**:

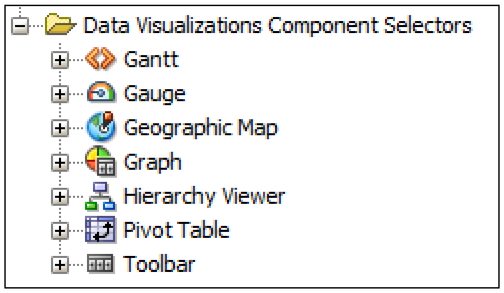


af|inputText:disabled::content
Name: hello

Background Color: #777777
Background Image: none
Content:
Color: #111111
Height: auto %

Data Visualizations Component Selectors

Finally, the **Data Visualizations Component Selectors** define the look of the various data visualization components (Gantt charts, graphs, maps, and so on):



Finding the selector at runtime

If you cannot find the selector you want in the skin editor, you can create a simple JSF page in JDeveloper and drop an instance of the component you want to skin onto this page and run the page in Firefox. Then start the Firebug add-on and inspect the element you want to skin.

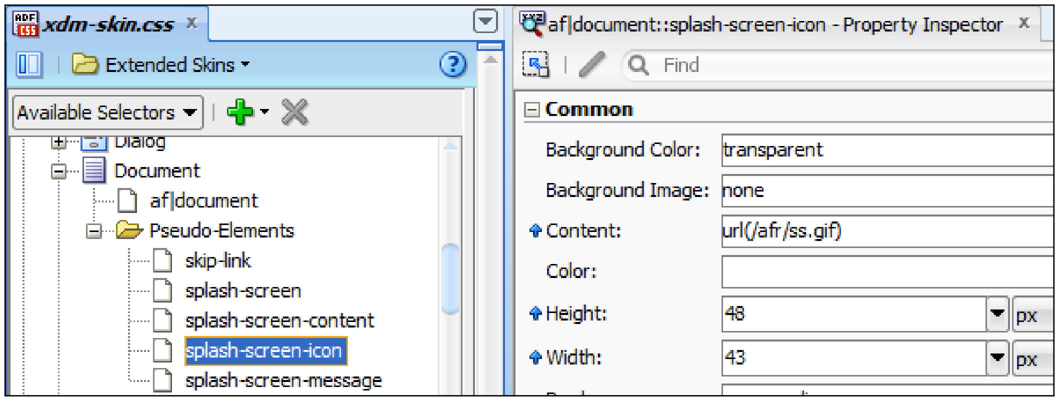
The right-hand panel in Firebug shows the styling that is applied to that element—if you set `DISABLE_CONTENT_COMPRESSION` to `true` as described earlier in this chapter, you will see a style class name such as `.af_panelFormLayout_label-cell`. This translates into the `af|panelFormLayout` component and the `label-cell` pseudo-element. You can then look this up in the skin editor and define the appearance you want.

Providing images for your skin

One of the things people often ask when they see their first ADF demo is: *Can I change the page loading image?* It seems that the blue spinning oval is not to everyone's liking.

Fortunately, you can change it. Unfortunately, the documentation does not list the images separately so you can see what you might want to override. However, in the skin selectors documentation (`http://jdevadf.oracle.com/adf-richclient-demo/docs/skin-selectors.html` or Google ADF skinning selectors), you will find subheadings **Icon Selectors** under the individual components, and these list the pseudo-element you need to style.

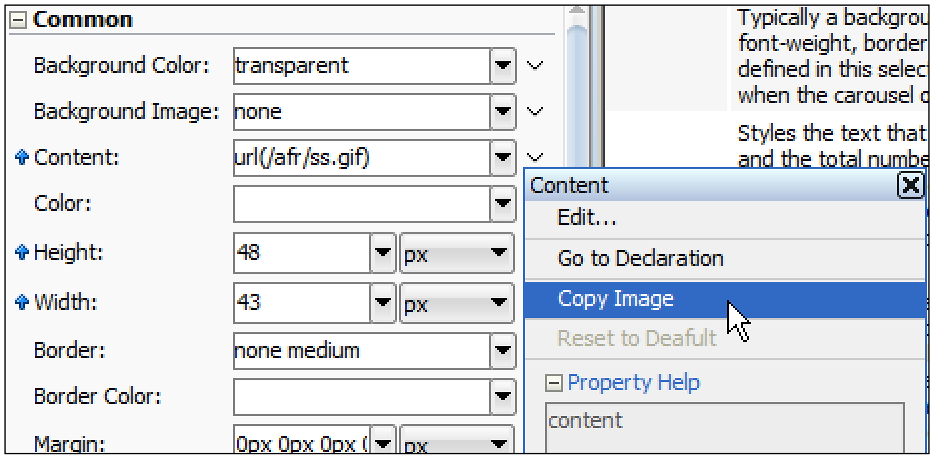
The page shown while the application loads (where you see the big spinning "O") is called the **splash page** and is styled under `af:document`. The icon is controlled by pseudo-element **::splash-screen-icon**:



Background Color: transparent
Background Image: none
Content: url(/afr/ss.gif)
Color:
Height: 48 px
Width: 43 px

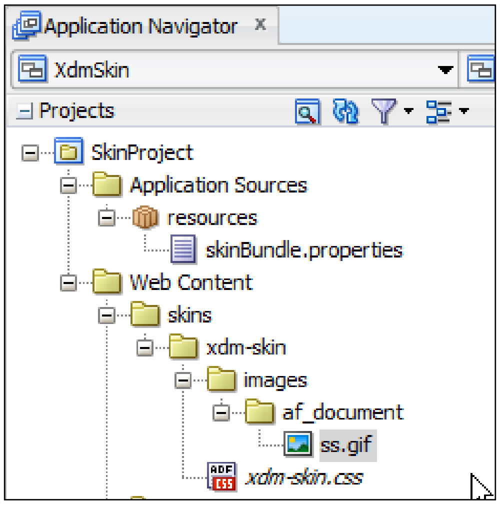
If you cannot find an image in the documentation, you can create a page containing the image, run it in Firefox and use Firebug to point to the image. The Firebug style window will show you the style that causes the image to be displayed.

Once you know which component and pseudo-element you want to style, select it in the skin editor, click on the little triangle next to the **Content** attribute and choose **Copy Image**:


Common
Background Color: transparent
Background Image: none
Content: url(/afr/ss.gif)
Color:
Height: 48 px
Width: 43 px
Border: none medium
Border Color:
Margin: 0px 0px 0px (px

Typically a backgrou
font-weight, border
defined in this selec
when the carousel c
Styles the text that
and the total numbe

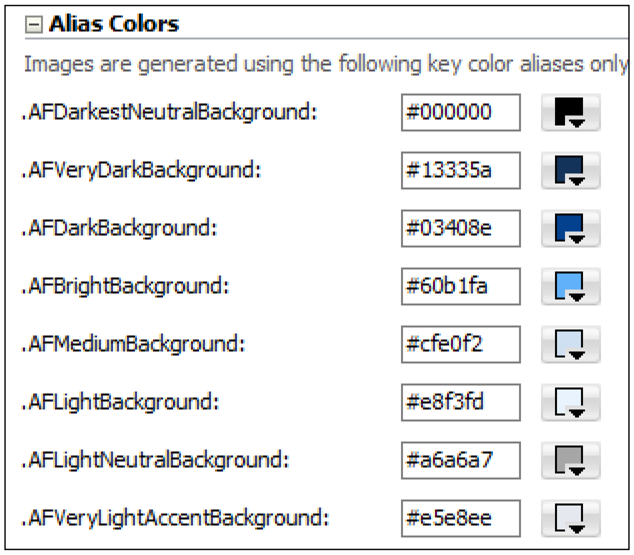
This places a copy of the image in your project—you can copy your own image into the same directory and change the **Content** attribute, or simply overwrite the standard image with your own:



Changing the color scheme

Changing the color scheme for a skin used to be a major undertaking, because ADF uses images everywhere a component has a rounded corner or a gradient background. If you needed to change the color scheme, you would have to change all of these files. But with the skin editor, this has become much easier.

To change the color scheme for your application, you click on the **Images** tab at the bottom of the skin editor panel. This tab allows you to change about a dozen base colors and generate all the necessary images to change your color scheme:

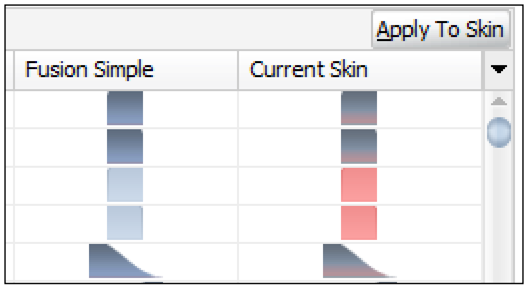


⊟ **Alias Colors**

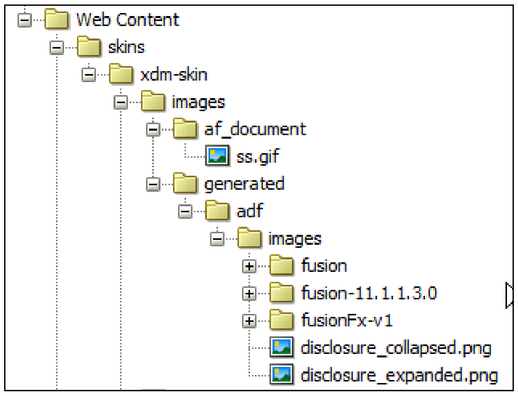
Images are generated using the following key color aliases only

.AFDarkestNeutralBackground:	#000000
.AFVeryDarkBackground:	#13335a
.AFDarkBackground:	#03408e
.AFBrightBackground:	#60b1fa
.AFMediumBackground:	#cfe0f2
.AFLightBackground:	#e8f3fd
.AFLightNeutralBackground:	#a6a6a7
.AFVeryLightAccentBackground:	#e5e8ee

You simply select other colors for the various global selectors by clicking on the color chooser button. As you change the colors, you will see the skin editor briefly displaying a message **Generating images**. When the new images have been generated, you can see the original skin and your modified skin side by side:

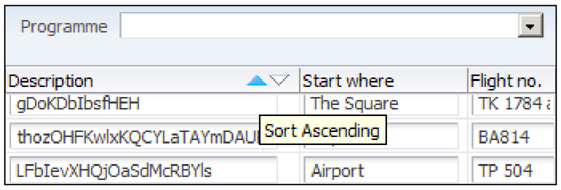


When you are happy with your color changes, you can click **Apply To Skin** to cause the skin editor to save all the generated images and refer to them in your ADF CSS file. In the **Application Navigator**, you can see all your newly generated image files under the **generated** node:



Creating a resource bundle for your skin

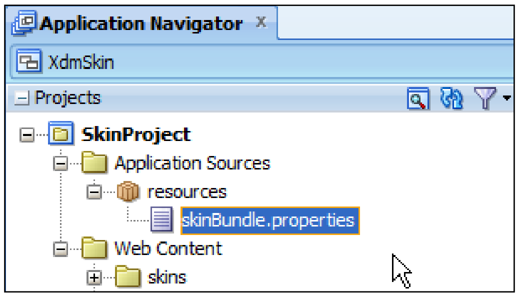
You might have noticed that many ADF components display texts that you cannot set through properties, for example, the pop-up help that appears if you allow sorting in an `af:table` component and point to the table header:



Description	Start where	Flight no.
gDoKDbIbsfHEH	The Square	TK 1784
thozOHFKwlxKQCYLaTAYmDAU		BA814
LFbIevXHQjOaSdMcRBYls	Airport	TP 504

If you want to change these standard strings, you first need to go to the documentation and find the **resource string** you want to override. For table sorting, you need `af_column.TIP_SORT_ASCENDING` and `af_column.TIP_SORT_DESCENDING`.

Then you need to open the resource bundle file that the skin editor has built for you; by default, it is called **skinBundle.properties**:

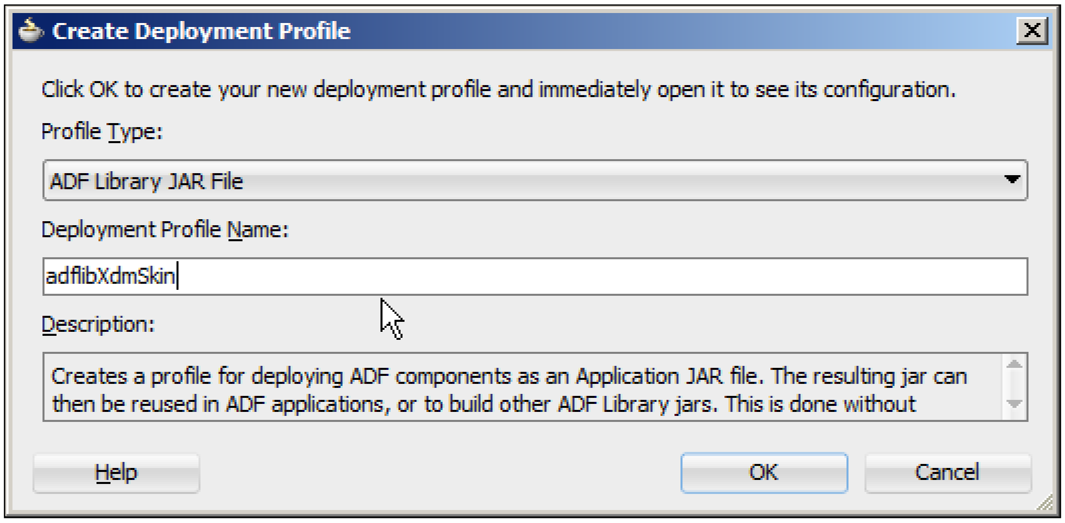


The default file contains a couple of examples, and you need to add your own resource strings to this file. It might look like the following:

```
# This file may be used to define alternative text
# for resource strings that appear in the user
# interface of ADF Components.
# Example: To change the text that appears on the
# buttons of the af:dialog component from Ok and
# Cancel to Continue and Go Back, add the following
# to this file:
#  AF_DIALOG.LABEL_OK = Continue
#  AF_DIALOG.LABEL_CANCEL = Go Back
AF_COLUMN.TIP_SORT_ASCENDING = First things first
AF_COLUMN.TIP_SORT_DESCENDING = The last shall be first
```

Packaging the skin

Once you are done with your skin, you need to package it into an ADF Library. To do this, you right-click on your *Skin* project and choose **Deploy | New Deployment Profile**. Choose **ADF Library JAR File** and give your deployment profile a name (for example, **adflibXdmSkin**):



Click OK to create your new deployment profile and immediately open it to see its configuration.

Profile Type: ADF Library JAR File

Deployment Profile Name: adflibXdmSkin

Description: Creates a profile for deploying ADF components as an Application JAR file. The resulting jar can then be reused in ADF applications, or to build other ADF Library jars. This is done without

In the **Edit ADF Library JAR Deployment Profile Properties** dialog that appears next, simply click **OK**.

Then right-click on your *Skin* project again and choose the name of your deployment profile (for example, **adflibXdmSkin**), click **Next** and then **Finish**. Like for other ADF libraries, this creates a JAR file in the `deploy` directory under your project. You, as developer, should add it to your version control system, and your build/deployment manager will pick it up, have it tested, and distributed to the various subsystem teams. If you are using a skin based on `fusionFx-simple` in JDeveloper 11g Rel. 1, you need to add `adf-richclient-fusion-simple_ps3.jar` (from the skin editor) to projects using your skin.

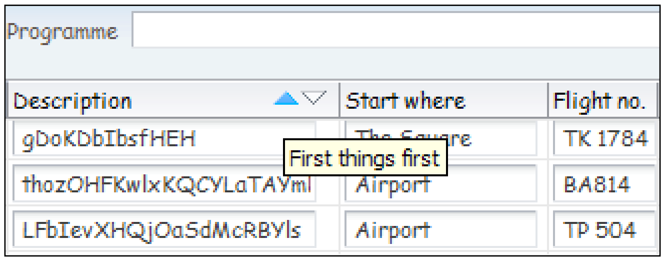
Using the skin

To use the skin, you simply need to add the ADF Library to your project from the **Component Palette** and change the `trinidad-config.xml` file in the project using the skin to refer to the skin in the library.

In the `trinidad-config.xml` file under **Web Content/WEB-INF** in your project, you need to change the `<skin-family>` tag to refer to the skin family you defined:

```
<skin-family>xdm-skin</skin-family>
```

If you now run your application, you should see your skinning take effect:



Programme

Description	Start where	Flight no.
gDoKDbIbsfHEH	The Square	TK 1784
thozOHFKwlxKQCYLaTAYml	Airport	BA814
LFbIevXHQjOaSdMcRBYls	Airport	TP 504

First things first

Summary

In this chapter, you have seen how ADF uses Cascading Style Sheets (CSS) for defining the appearance of components without affecting their functionality. For changing the look of an individual component, you can use inline styles, content styles, and style classes.

If you want to customize the look of the entire application, you define a skin. This used to be difficult and complex, but with the skin editor available both integrated in JDeveloper and as a stand-alone product, this has become much easier. You have a tree navigator for selecting components, you can use the **Property Inspector** to change settings and immediately see what your component will look like. Your final skin can include both global changes affecting the whole application, including the color scheme, and visual changes that affect only one specific component or even just one aspect of it.

When you are done with your application skin, you can deploy it as an ADF library using the normal procedures for working with ADF libraries to use it in your subsystems and master application.

Your manager is impressed with the way you can easily customize the look of the application and asks you if you can also customize the functionality. Yes, you say, using the ADF customization features. These will be the topic of the next chapter.

9
Customizing the Functionality

You have seen that you can change the way your application looks—and that you can deploy the same application to different users with different visual appearance. But your boss at DMC Solutions would like to be able to sell the XDM application to different customers, offering each of them a slightly different functionality.

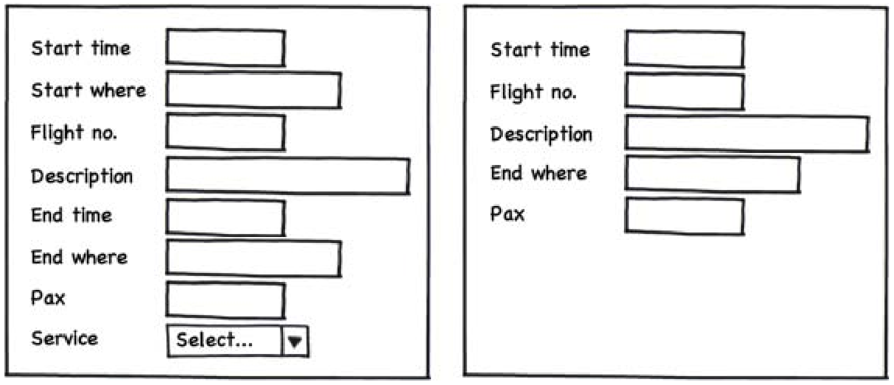
Fortunately, ADF makes that easy through a feature known as **customization**.

Why customization?

The reason ADF has customization features built-in is because Oracle Fusion Applications need them. Oracle Fusion Applications is a suite of programs capable of handling every aspect of a large organization—personnel, finance, project management, manufacturing, logistics, and much more. Because organizations are different, Oracle has to offer a way for each customer organization to fit Oracle Fusion Applications to their requirements.

This customization functionality can also be very useful for organizations that do not use Oracle Fusion Applications. If you have two screens that work with the same data, but one of the screens must show more fields than the other, you can create one screen with all the fields and use customization to create another version of the same screen with fewer fields for other users.

For example, the destination management application that we are using as an example in this book might have a data entry screen showing all details of a task to a dispatcher, but only the relevant details to an airport transfer guide:



Companies such as DMC Solutions that produce software for sale realize an additional benefit from the customization features in ADF. DMC Solutions can build a base application, sell it to different customers and customize each instance of the application to that customer without changing the base application.

How does an ADF customization work?

More and more Oracle products are using something called **Meta Data Services** to store metadata. Metadata is data that describes other pieces of information—where it came from, what it means, or how it is intended to be used. An image captured by a digital camera might include metadata about where and when the picture was taken, which camera settings were used, and so on. In the case of an ADF application, the metadata describes how the application is intended to be used.

There are three kinds of customizations in ADF:

- **Seeded customizations**: They are customizations defined in advance (before the user runs the application) by customization developers.
- **User customizations** (sometimes called *personalizations*): They are changes to aspects of the user interface by application end users. The ADF framework offers a few user customization features, but you need additional software such as Oracle WebCenter for most user customizations. User customizations are outside the scope of this book.

- **Design time at runtime**: They are advanced customization of the application by application administrators and/or properly authorized end users. This requires that application developers have prepared the possible customizations as part of application development—it is complicated to program using only ADF, but Oracle WebCenter provides advanced components that make this easier. This is outside the scope of this book.

Your customization metadata is stored in either files or a database repository. If you are only planning to use seeded customizations, a file-based repository is fine. However, if you plan to allow user customizations or design time at runtime, you should set up your production server to store customizations in a metadata database. Refer to the *Fusion Middleware Administrator's Guide* for information about setting up a metadata database.

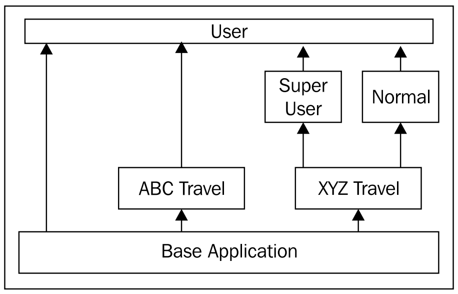
Applying the customization layers

When an ADF application is customized, the ADF framework applies one or more **customization layers** on top of the **base application**. Each layer has a value, and customizations are assigned to a specific customization layer and value.

The concept of multiple layers makes it possible to apply, for example:

- Industry customization (customizing the application for example, the travel industry: `industry=travel`)
- Organization customization (customizing the application for a specific travel company: `org=xyztravel`)
- Site customization (customizing the application for the Berlin office)
- Role-based customization (customizing the application for casual, normal, and advanced users)

The XDM application that DMC Solution is building could be customized in one way for ABC Travel and in another way for XYZ Travel, and XYZ Travel might decide to further customize the application for different types of users:



You can have as many layers as you need—Oracle Fusion Applications is reported to use 12 layers, but your applications are not likely to be that complex.

For each customization layer, the developer of the base application must provide a **customization class** that will be executed at runtime, returning a value for each customization layer. The ADF framework will then apply the customizations that the customization developer has specified for that layer/value combination. This means that the same application can look in many different ways, depending on the values returned by the customization classes and the customizations registered:

Org layer value	Role layer value	Result
`qrstravel`	`any`	Base application, because there are no customizations defined for QRS Travel
`abctravel`	`any`	The application customized for ABC Travel, because there are no role layer customizations for ABC Travel, the value of the role layer does not change the application
`xyztravel`	`normal`	The application customized for XYZ Travel and further customized for normal users in XYZ Travel
`xyztravel`	`superuser`	The application customized for XYZ Travel and further customized for super users in XYZ Travel

Making an application customizable

To make an application customizable, you need to do three things:

1. Develop a customization class for each layer of customization.
2. Enable seeded customization in the application.
3. Link the customization class to the application.

The customization developer, who will be developing the customizations, will additionally have to set up JDeveloper correctly so that all customization levels can be accessed. This setup is described later in the chapter.

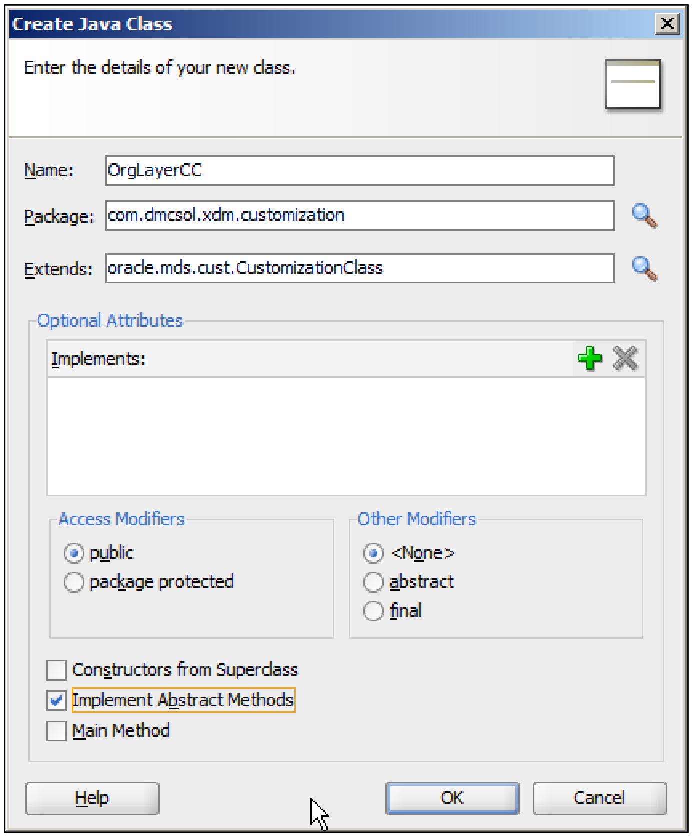
Developing the customization classes

For each layer of customization, you need to develop a customization class with a specific format—technically, it has to extend the Oracle-supplied abstract class `oracle.mds.cust.CustomizationClass`.

A customization class has a name (returned by the `getName()` method) and a value (returned by the `getValue()` method). At runtime, the ADF framework will execute the customization classes for all layers to determine the customization value at each level. Additionally, the customization class has to return a short unique prefix to use for all customized items, and a cache hint telling ADF if this is a static or dynamic customization.

Building the classes

Your customization classes should go in your Common Code workspace. A customization class is a normal Java class, that is, it is created with **File** | **New** | **General** | **Java Class**. In the **Create Java Class** dialog, give your class a name (`OrgLayerCC`) and place it into a customization package (for example, `com.dmcsol.xdm.customization`). Choose to extend **oracle.mds.cust.CustomizationClass** and check the **Implement Abstract Methods** checkbox:



Name: OrgLayerCC
Package: com.dmcsol.xdm.customization
Extends: oracle.mds.cust.CustomizationClass

Create a similar class called `RoleLayerCC`.

Implementing the methods

Because you asked the JDeveloper to implement the abstract methods, your classes already contain three methods:

- `getCacheHint()`
- `getName()`
- `getValue(RestrictedSession, MetadataObject)`

The `getCacheHint()` method must return an `oracle.mds.cust.CacheHint` constant that tells ADF if the value of this layer is static (common for all users) or dynamic (depending on the user). The normal values here are `ALL_USERS` for static customizations or `MULTI_USER` for customizations that apply to multiple users. In the XDM application, you will use:

- `ALL_USERS` for `OrgLevelCC`, because this customization layer will apply to all users in the organization
- `MULTI_USER` for `RoleLevelCC`, because the role-based customization will apply to multiple users, but not necessarily to all

Refer to the chapter on customization with MDS in *Fusion Developer's Guide for Oracle Application Development Framework* for information on other possible values.

The `getName()` method simply returns the the name of the customization layer.

The `getValue()` method must return an array of `String` objects. It will normally make most sense to return just one value—the application is running for exactly one organization, you are either a normal user or a super user. For advanced scenarios, it is possible to return multiple values, in such a case multiple customizations will be applied at the same layer. Each customization that a customization developer defines will be tied to a specific layer and value—there might be a customization that happens when `org` has the value `xyztravel`.

For the `OrgLayerCC` class, the value is static and is defined when DMC Solutions installs the application for XYZ Travel—for example, in a property file. For the `RoleLayerCC` class, the value is dynamic, depending on the current user, and can be retrieved from the ADF security context. The `OrgLayerCC` class could look like the following:

```
package com.dmcsol.xdm.customization;

import …

public class RoleLayerCC extends CustomizationClass {
public CacheHint getCacheHint() {
    return CacheHint.MULTI_USER;
  }

  public String getName() {
    return "role";
  }

  public String[] getValue(RestrictedSession restrictedSession,
MetadataObject metadataObject) {
    String[] roleValue = new String[1];
    SecurityContext sec = ADFContext.getCurrent().
getSecurityContext();
    if (sec.isUserInRole("superuser")) {
      roleValue[0] = "superuser";
    } else {
      roleValue[0] = "normal";
    }
    return roleValue;
  }
}
```

The `GetCacheHint()` method returns `MULTI_USER` because this is a dynamic customization—it will return different values for different users.

The `GetName()` method simply returns the name of the layer.

The `GetValue()` method uses `oracle.adf.share.security.SecurityContext` to look up if the user has the super user role and returns the value `superuser` or `normal`.

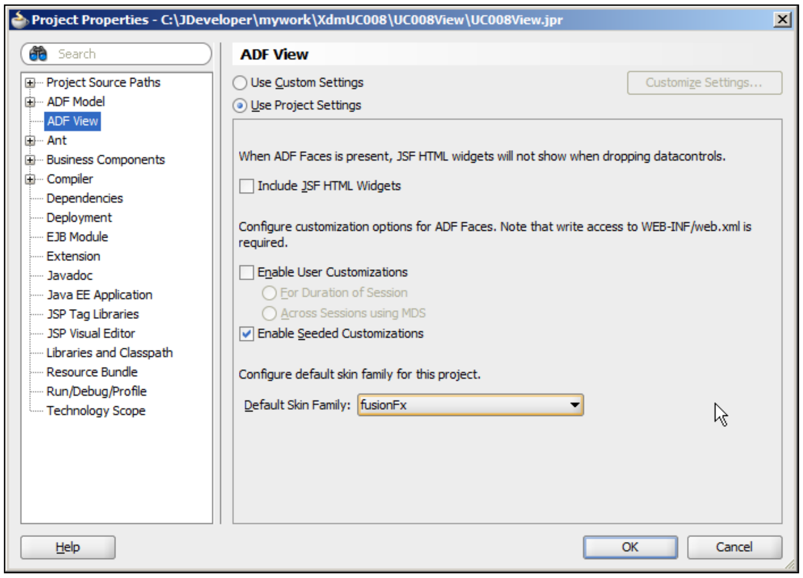
Deploying the customization classes

Because you place your customization class in the Common Code project, you need to deploy the Common Code project to an ADF library and have the build/configuration manager copy it to your common library directory.

Enabling seeded customization

In order to be able to customize pages, you need to tell JDeveloper to prepare for customization as you add components to the page flows and pages in your *ViewController* project.

In this chapter, we will customize the *UC008* subsystem. Open the `XdmUC008` workspace and choose the **Project Properties** dialog for the View project (**UC008View**) in the workspace for each task flow you want to customize. Choose **ADF View** and then check the checkbox **Enable Seeded Customizations**:


Project Properties - C:\JDeveloper\mywork\XdmUC008\UC008View\UC008View.jpr

ADF View

[ ] Use Custom Settings
[✓] Use Project Settings

When ADF Faces is present, JSF HTML widgets will not show when dropping datacontrols.

[ ] Include JSF HTML Widgets

Configure customization options for ADF Faces. Note that write access to WEB-INF/web.xml is required.

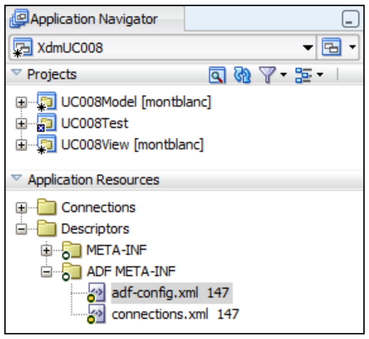
[ ] Enable User Customizations
( ) For Duration of Session
( ) Across Sessions using MDS
[✓] Enable Seeded Customizations

Configure default skin family for this project.

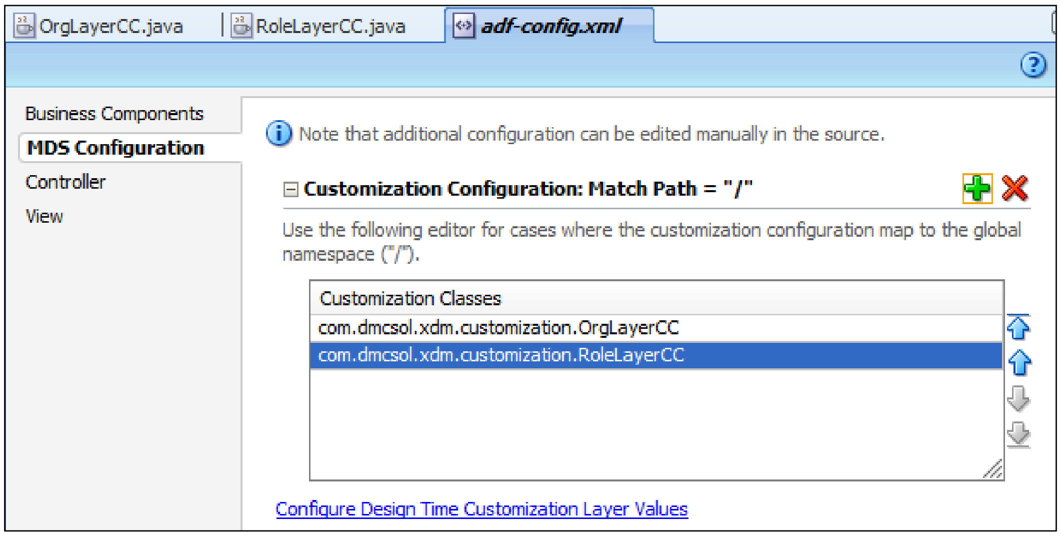
Default Skin Family: fusionFx

Linking the customization class to the application

The last step in preparing an application for customization is to define which customization levels you will use, and in which order they will be applied. This is done in the `adf-config.xml` file that is found under the **Application Resources** panel in the **Application Navigator**. Expand the **Descriptors** node and then **ADF META-INF** to find the **adf-config.xml** file:



Open this file and choose the **MDS Configuration** sub-tab. On this tab, you can add all of your customization classes and define the order in which they are applied (the top-most class in this dialog is applied first):



Note that additional configuration can be edited manually in the source.

Customization Configuration: Match Path = "/"

Use the following editor for cases where the customization configuration map to the global namespace ("/").

Customization Classes
com.dmcsol.xdm.customization.OrgLayerCC
com.dmcsol.xdm.customization.RoleLayerCC

Your customization classes defined in the Common Code workspace are not available until you have deployed the Common Code workspace to an ADF library and have made this library available to the project where you want to perform customization.

Configuring the customization layers

When developing customizations, the customization developer must choose a layer and a value to work in—for example, developing customizations for `org = xyztravel`. But as the layer values are returned by code at runtime, there is no way for JDeveloper to figure out which layer values are available. That is why you need to define the available customization layers and the possible values in a `CustomizationLayerValues.xml` file.

When you are developing customizations, JDeveloper will read the `adf-config.xml` file to determine which customization classes to use, and the `CustomizationLayerValues.xml` file to determine which layer values should be made available for the developer to select from. The static `CustomizationLayerValues.xml` file must contain all the layers defined in `adf-config.xml`, and should contain all the possible values your customization class can return.

Match the XML file with the customization class

The `CustomizationLayerValues.xml` values are used at design time, and the customization class is called at run time. If you leave out a value from the XML file, you will not be able to define customizations for that value. If you add a value to the XML file that will never be returned by the customization class, you might accidentally define a customization that will never be applied at run time.

To edit the `CustomizationLayerValues.xml` file, click on the **Configure Design Time Customization Layer Values** link at the bottom of the **MDS Configuration** tab in the `adf-config.xml` file. The first time you edit the file, you get a warning about overriding the global `CustomizationLayerValues.xml` file. Simply acknowledge this warning and continue to your file.

You are presented with an example file containing some customization layer values and descriptive comments. For the XDM application using two customization layers of `org` and `role`, change the file to look like the following:

```
<cust-layers xmlns="http://xmlns.oracle.com/mds/dt">
  <cust-layer name="org" id-prefix="o"  value-set-size="small">
    <cust-layer-value value="abctravel" display-name="ABC Travel"
      id-prefix="a"/>
    <cust-layer-value value="xyztravel" display-name="XYZ Travel"
      id-prefix="x"/>
  </cust-layer>
  <cust-layer name="role" id-prefix="r" value-set-size="small">
    <cust-layer-value value="normal" display-name="Normal user"
      id-prefix="n"/>
    <cust-layer-value value="superuser" display-name="Superuser"
      id-prefix="s"/>
  </cust-layer>
</cust-layers>
```

The `name` attribute of the `cust-layer` element must match the values returned from the `getName()` methods in your customization class. Similarly, the `value` attribute of the `cust-layer-value` element must match the values returned by the `getValue()` method in the corresponding customization class.

The `id-prefix` values for both layer and layer value are used to ensure that each component is given a globally unique name. All components in an ADF Faces application are given a component ID—for example, an `inputText` element might get an ID such as `it3` in the base application. With customization, however, that same element might exists in many different variations—customized in one way for one organization and in another way for another organization. To ensure that ADF can tell all of these components apart, it adds prefixes for all layers to the ID of the component as specified in this file, for example:

- The input component in the base application would be `it3`
- The same component customized in the `org` layer for the value `abctravel` would be `oait3` (the `o` from the `org` ID-prefix and the `a` from the `abctravel` ID-prefix)
- The same component customized in the `org` layer for the value `xyztravel` and in the `role` layer for the value `superuser` would be `oxrsit3` (`o` from org layer, `x` from `xyztravel` value, `r` from `role` layer, `s` from `superuser` value)

The `display-name` is shown to the customization developer to help select the right layer, and the `value-set-size` parameter should normally be set to `small` (this causes JDeveloper to show a drop-down box for selecting layer value).

Setting up JDeveloper for customization

To perform the actual customization of an application, you need to set up JDeveloper to understand the available customization layers and classes, and you need to run JDeveloper in the special **Customization Developer** role.

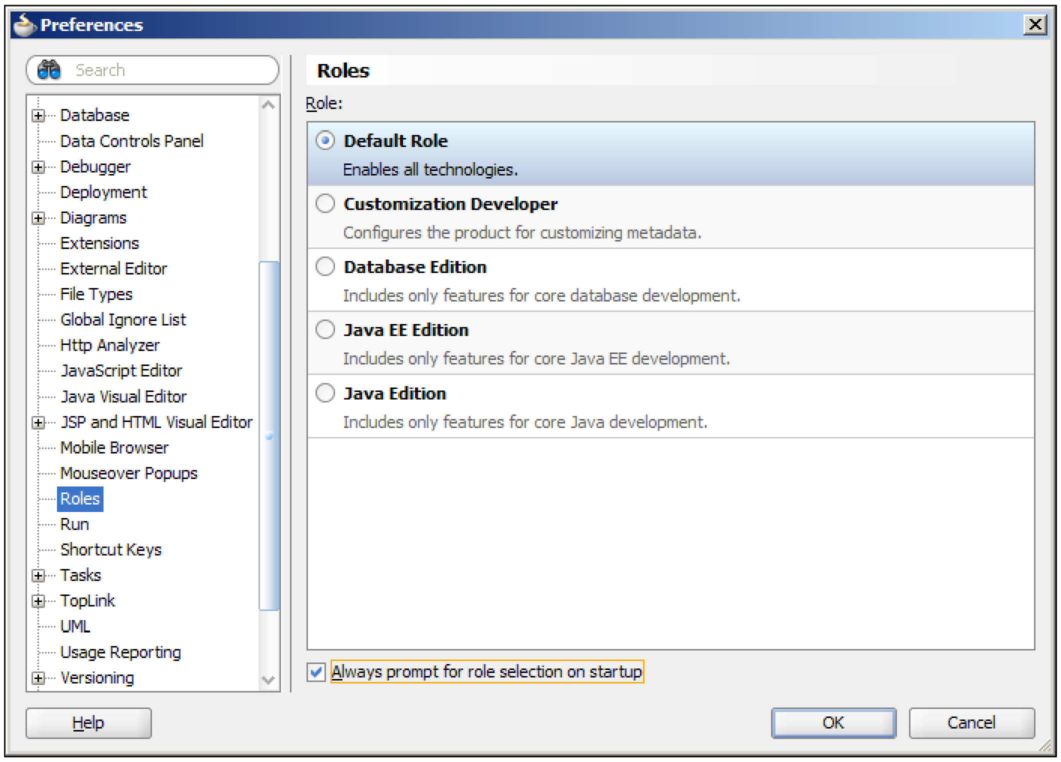
Making the customization class available to JDeveloper

First, the customization class must be placed so that JDeveloper can always find it (technically, the customization class has to be on JDeveloper's **classpath**). This is necessary because while you are developing, your application is not necessarily running. This means that there might not be an instance of the customization class available for JDeveloper to call.

To make your customization class available to JDeveloper at all times, you need to copy the JAR file containing it to a directory that is on the JDeveloper classpath. For the XDM project, copy the `adflibCommonCode.jar` file from `C:\JDeveloper\XdmLib` to the ...`\jdeveloper\jdev\lib\patches` directory.

Selecting the customization role

The first time you started JDeveloper, you were greeted with a role selection dialog. You have probably deselected it by now, but you can re-enable it by choosing **Tools** | **Preferences** | **Roles** and check the checkbox **Always prompt for role selection on startup**:



Roles

Role:

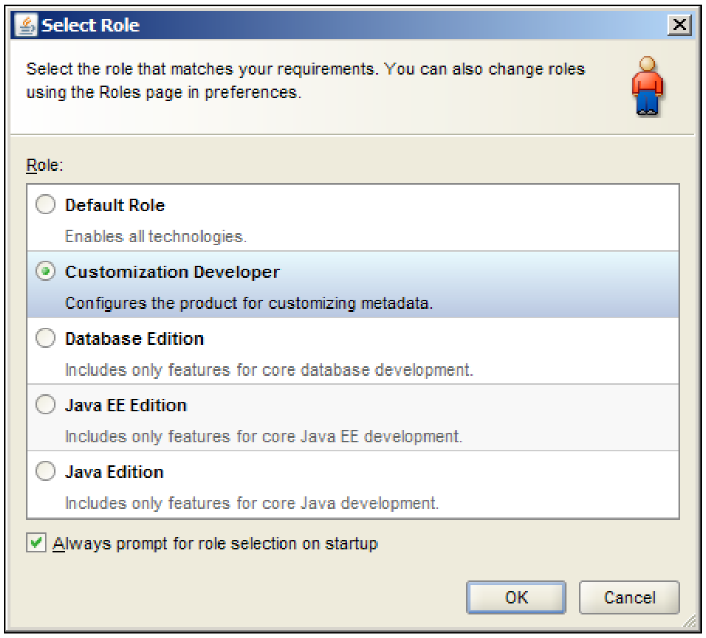
- (●) **Default Role** — Enables all technologies.
- ( ) **Customization Developer** — Configures the product for customizing metadata.
- ( ) **Database Edition** — Includes only features for core database development.
- ( ) **Java EE Edition** — Includes only features for core Java EE development.
- ( ) **Java Edition** — Includes only features for core Java development.

[✓] Always prompt for role selection on startup

The next time you start JDeveloper, you will be prompted for role selection again.

Performing the customization

With your application prepared for customization and JDeveloper set up correctly, you are ready to perform the actual customization. Because you cannot change developer role while JDeveloper is running, you need to exit JDeveloper and start it again. You will be prompted to select a role—choose **Customization Developer**:



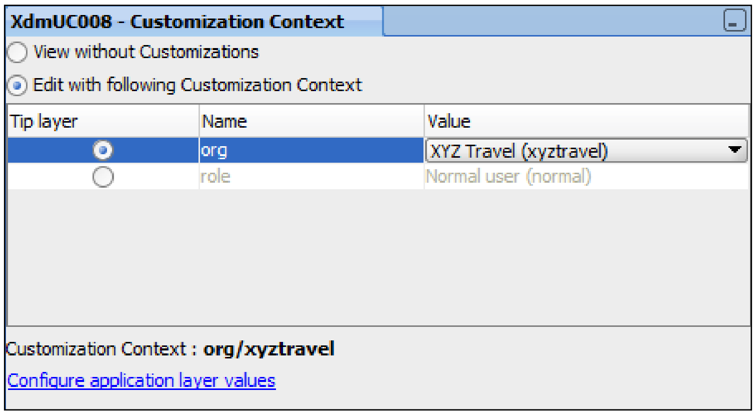
Select the role that matches your requirements. You can also change roles using the Roles page in preferences.

Role:

- Default Role — Enables all technologies.
- [✓ Customization Developer] — Configures the product for customizing metadata.
- Database Edition — Includes only features for core database development.
- Java EE Edition — Includes only features for core Java EE development.
- Java Edition — Includes only features for core Java development.

[✓] Always prompt for role selection on startup

When JDeveloper is running in customization mode, you will notice the new **Customization Context** panel shown at the bottom of the screen, next to the usual **Log** window:



View without Customizations
[✓ Edit with following Customization Context]

Tip layer	Name	Value
●	org	XYZ Travel (xyztravel)
○	role	Normal user (normal)

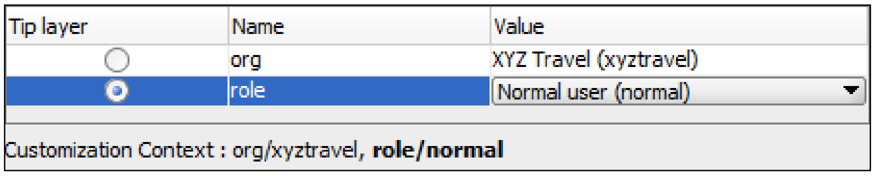
Customization Context : **org/xyztravel**

Configure application layer values

The values shown in this window come from the `CustomizationLayerValues.xml` file you created—the values in the **Name** column corresponds to the `cust-layer` elements and the values in the **Value** column correspond to the `cust-layer-value` elements.

When developing customizations, you are always working on a specific layer, called the **tip layer**. You select the tip layer with the radio buttons in the **Tip layer** column in the **Customization Context** panel as shown previously. In the preceding illustration, the **org** layer is selected as the tip layer and the **Value** is **XYZ Travel**. This means that the customizations you make will be registered to the customization context **org/xyztravel**.

You see that the **role** layer value is grayed out when **org** is the tip layer. Because the **org** layer is below the **role** layer, it does not matter what the value of **role** is when you are customizing for the organization. When you select **role** as tip layer, the **Value** column becomes active:



Tip layer	Name	Value
○	org	XYZ Travel (xyztravel)
●	role	Normal user (normal)

Customization Context : org/xyztravel, **role/normal**

Any customizations that you register with the preceding settings will be registered in the role layer in the customization context **org/xyztravel, role/normal**.

If you want to see the base application, select **View without Customizations**.

Customizing business components

You can customize some aspects of business components—for example, it is possible to add or change validation rules on entity objects. This can be useful for example if XYZ Travel wants to change a validation rule in the base application, but ABC Travel is happy with the base application functionality.

In this case, you select **org** as the tip layer and select **XYZ Travel** as **Value**, and change the validation rule as XYZ Travel requires. The base application remains unchanged, but JDeveloper has now stored a customization in the context `org/xyztravel`. When XYZ Travel is running the application, the `GetValue()` method in the `OrgLayerCC` class will return `xyztravel`, the customization is applied and the modified business rule applies. When ABC Travel is running the application, the `GetValue()` method will return `abctravel`, and the unmodified validation rule from the base application is applied.

You can also customize UI hints (for example, default labels) by assigning new resource strings to them as described in a later section in this chapter.

Customizing the pages

You can customize your pages and page fragments in many ways:

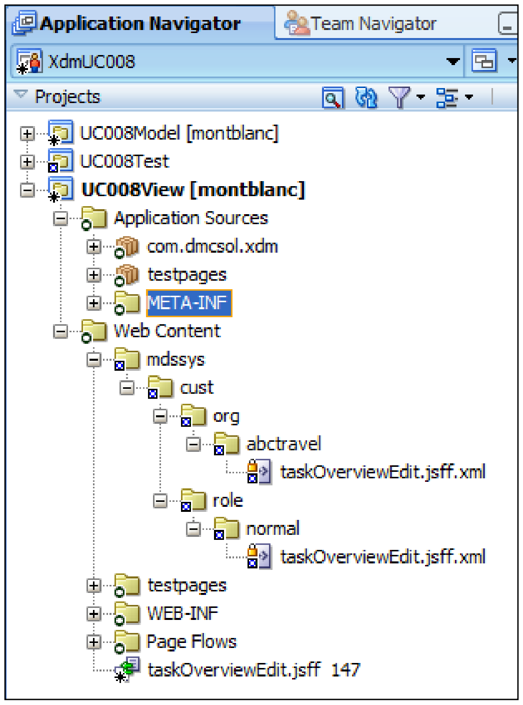
- You can add new fields, buttons, and layout components
- You can remove fields
- You can reorder existing items—reorder fields, move fields to other layout components, and so on
- You can assign new resource strings to components

In order to make a change, you have to work in the **Design** view and the **Property Palette**. You can change to the **Source** view of your page, but the source is read-only when running JDeveloper in the **Customization Developer** mode.

As a simple example, imagine that ABC Travel does not require the *Flight No.* column and want the *StartWhere* column before the *Text* column. Additionally, normal users should not be shown the *Comment* column.

To perform this customization, first choose **org** as the **Tip layer** and **ABC Travel** as the **Value**. Then open the **taskOverviewEdit.jsff** page fragment and delete the **FlightNo** column and move the **StartWhere** column. Then you choose **role** as the **Tip layer**, leave **ABC Travel** selected as **Value** for **org** and select **Normal user** as **Value** for **role**. The JDeveloper will show a message that it has to close customizable files. Click **OK**, re-open the **taskOverviewEdit.jsff** page fragment and delete the **Comment** column.

While you were performing this customization, JDeveloper was storing your customizations as XML files with the same name as the base object. After the preceding customization, your application navigator will now show a couple of new files:



The **taskOverviewEditPage.jsff.xml** file shown under **mdssys/cust/org/abctravel** contains the customizations that you defined for ABC Travel, and file under **mdssys/cust/role/normal** contains the the customizations that you defined for normal users. These files record the changes you have made in a compact XML format, as follows:

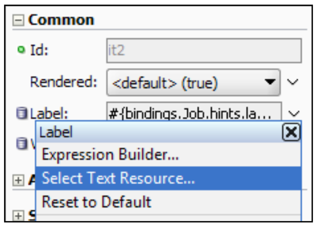
```
<mds:customization version="11.1.1.59.23"
                   xmlns:mds="http://xmlns.oracle.com/mds">
  <mds:replace node="c8"/>
  <mds:move node="c4" after="c2"/>
</mds:customization>
```

If you want to see your unmodified base application, you can select **View without Customizations** in the **Customization Context** panel.

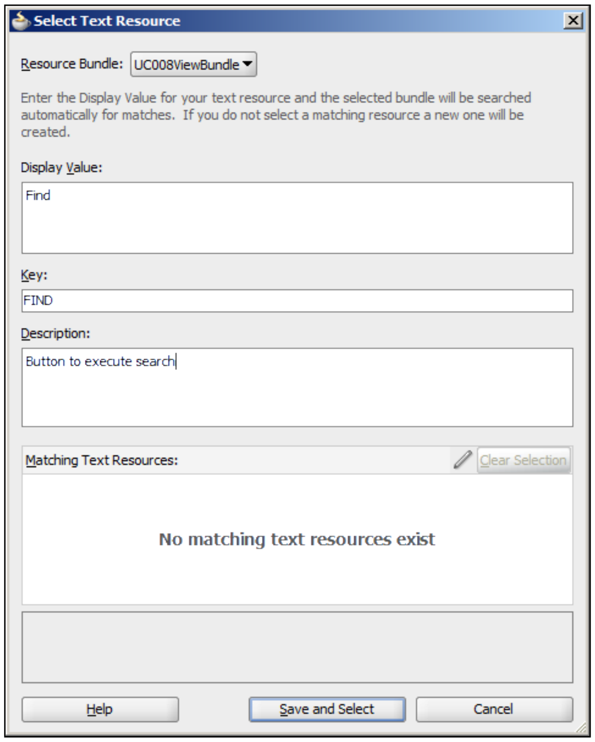
Customizing strings

When customizing an application, you will find that the attributes defining the text displayed to the users are grayed out—you cannot change the **Text** property of a button or the **Label** property of a field. What you can do, however, is to assign new **resource strings** to items.

Instead of just setting the **Text** property for a button to a literal value like **Cancel**, you can click on the little down triangle to the right of the property field and choose **Select Text Resource** to bring up the **Select Text Resource** dialog:



Alternatively, you can invoke the **Select Text Resource** dialog by right-clicking on a field and choosing **Select Text Resource for | Label**:



Resource Bundle: UC008ViewBundle

Enter the Display Value for your text resource and the selected bundle will be searched automatically for matches. If you do not select a matching resource a new one will be created.

Display Value: Find

Key: FIND

Description: Button to execute search

Matching Text Resources:

No matching text resources exist

In this dialog, you can write a new text in the **Display Value** field or select an existing UI string from the **Matching Text Resources** box.

If you add a new string while in **Customization Developer** mode, JDeveloper will ask you to confirm that you want to add a new key to the **override bundle**. Your base application already has a resource bundle, but the changes you make while customizing is stored in a separate resource bundle that the JDeveloper automatically creates.

Your customizations are stored just like other customizations. If you are customizing for ABC Travel and change the **Search** button **Text** attribute on the the **taskOverviewEditPage.jsff** page fragment, the **taskOverviewEditPage.jsff.xml** file under **mdssys/cust/org/abctravel** will be updated to include something like the following:

```
<mds:modify element="cb1">
  <mds:attribute name="text" value="#{uc008viewBundle.FIND}"/>
</mds:modify>
```

The actual override bundle containing your custom strings and their keys is not displayed in the JDeveloper. You can find it in the file system in the subdirectory `resourcebundles\xliffBundles` under the root directory of your application. For example, the override bundle for `XdmUC008` might be found in `C:\JDeveloper\mywork\XdmUC008\resourcebundles\xliffBundles\XdmUC008OverrideBundle.xlf`.

This file is a normal XLIFF file like you might be using in your base application. If you only defined one new string `Find`, it would look like the following:

```
<?xml version="1.0" encoding="windows-1252" ?>
<xliff version="1.1" xmlns="urn:oasis:names:tc:xliff:document:1.1">
  <file source-language="en" original="this" datatype="x-oracle-adf">
    <body>
      <trans-unit id="FIND">
        <source>Find</source>
        <target/>
        <note>Button to execute search</note>
      </trans-unit>
    </body>
  </file>
</xliff>
```

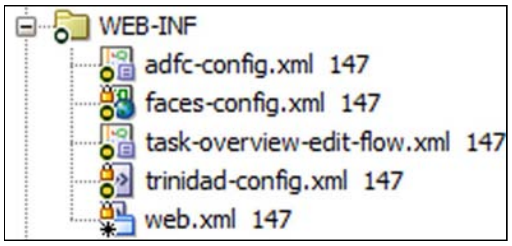
There is more information on using resource bundles in *Appendix A, Internationalization*.

What cannot be customized?

The ADF customization works by registering changes to the XML files in other XML files.

This means that you cannot customize Java code—you have to develop all your Java code in the base application. But as you can customize the attributes, it is possible to customize your application to select among different methods from the base application. For example, a button has an **ActionListener** property that can point to a method binding. If you have implemented two different methods in the base application and have created action bindings for them, you can use customization to choose one or the other.

You also cannot customize certain configuration files, resource bundles or security-related files. When you are running JDeveloper in the **Customization Developer** mode, the **Application Navigator** will show all of these with a small padlock icon in the top-left corner of the item icon. In the following example, you can see that the **faces-config.xml**, **trinidad-config.xml**, and **web.xml** files cannot be customized:



```
WEB-INF
  adfc-config.xml 147
  faces-config.xml 147
  task-overview-edit-flow.xml 147
  trinidad-config.xml 147
  web.xml 147
```

Summary

The ADF customization features are just what an independent software vendor such as DMC Solutions need. The development team can build one base application, and implementation teams can customize the application for different customers in a controlled manner without affecting the base application.

You have seen how to prepare an application for customization and how to perform customization. You have also seen that your customizations are stored separately from the application so they can be re-used even if you provide a new version of the base application.

Even if your enterprise application is not intended to be implemented for different customers, you can still use customization features to offer different versions of screens to different users. The next chapter will explain how to secure your application.

10
Securing your ADF Application

Back in 2004, someone found out that you could open a high-security bicycle lock made out of hardened steel—with a ballpoint pen! (Google *bike lock ballpoint pen* for more information). This was an excellent product that would resist bolt cutters, hacksaws, and crowbars—but the circular lock had a glaring weakness.

Security is like that: Only as strong as the weakest link. To make certain that your enterprise ADF application is secure, you need security at many levels—your servers must be secure, your network must be secure, your application must be secure, and your data must be secure.

This chapter is only concerned with securing your enterprise ADF application using the many easy-to-use security features built into ADF. However, remember to consider the other aspects of security as well.

Security basics

Two important parts of security are **authentication** (determining who the user is) and **authorization** (determining what the user is allowed to do). As an ADF application is a standard Java EE application and runs inside a Java EE application server, it can make use of the security features of Java EE and does not have to implement everything itself.

Authentication

A JAVA EE application server offers to handle security for the applications that run inside it—so-called **container-managed security**. This approach offers several types of authentication—for an enterprise ADF application, you will always choose form-based authentication. This allows the application to point to a web page (a login form) where the user can enter her username and password. You can design this login page as part of your application so that it looks like the rest of the application.

Alternatives are *Basic* or *Digest* authentication; both of these depend on the browser to present a login screen. Basic even sends your password unencrypted, unless you use SSL. You might have seen this in use on basic websites. You will simply get a dialog box with two fields for username and password in whatever look the browser has decided to give it. You do not want this for your beautiful enterprise ADF application. Additionally, there is no way to log out of an application using Basic or Digest authentication short of closing the browser.

The actual verification of the username and password is handled by the Oracle WebLogic application server using an **authentication provider**. The authentication provider connects WebLogic with some identity store—WebLogic comes with pre-built providers for the built-in LDAP server, Oracle Internet Directory, Microsoft Active Directory, and relational databases. If none of these meet your needs, you can even program your own authentication provider. Setting up authentication providers is the responsibility of the application server administrator.

Authorization

While container-managed security works fine for authentication, the authorization features are a bit too basic for enterprise ADF applications. Container-managed authorization only protects specific URL patterns, but this is not enough for a modern, dynamic modern web application. JavaServer Faces uses server-side navigation, so the same URL can represent many different parts of the application. You saw an example of this in *Chapter 6*, *Building the Enterprise Application*, where we used a dynamic region to swap task flows within the same page. It is obvious that a simple protection of URLs is not enough to secure our ADF application.

To achieve a more fine-grained authorization control, we therefore turn to the **Java Authentication and Authorization Service** (**JAAS**). This technology offers to protect different resource objects—not just URL patterns. Unfortunately, JAAS-based web application authorization is not standardized across application servers, so an application using JAAS security is not portable across application servers.

The Oracle security solution

Because of these integration challenges, Oracle offers **Oracle Platform Security Services** (**OPSS**) as part of Oracle Fusion Middleware.

OPSS provides a common interface to authentication and authorization and handles the intricacies of the integration, and is used for ADF security as well as in many other Oracle products.

So, even though the underlying technology can be quite complex, the application developer is presented with a simple, secure solution: ADF Security.

But wait, there is more!

ADF Security makes it easy to build a secure application, but this one chapter cannot address every security issue. Every organization should appoint a security officer with the responsibility to ensure the correct level of security, considering your data, users, and threat environment.

Security decisions

The first security decision is whether you need to secure the application at all. As the focus of this book is enterprise applications, we assume that you do need to secure it. If you do not need security, feel free to skip to the next and final chapter on packaging and deploying your application.

Authentication

In an enterprise setting, you will normally already have an identity management infrastructure of some kind in place—Microsoft Active Directory, an LDAP-server such as Oracle Internet Directory, or some other solution. We do not want each application handling its own users, but want to integrate with the existing identity infrastructure. This means that all applications should delegate the *authentication* to the application server, and the application server must be integrated with the existing authentication mechanism.

Authorization

However, you cannot delegate the *authorization*. Only you, the application developer, knows what pages, task flows, and data elements need to be accessible to different categories of users. So as part of your application design, you must identify the **application roles**—logic groupings of users that you can give detailed access rights. You should be able to identify these roles from your application requirements; if not, you need to go back to your end users to determine the roles the application needs. For example, the destination management application we use in this book needs the roles `OperationsStaff`, `AdminStaff`, `EventResponsible`, `Finance`, and `Manager`.

Where to implement security

If you are structuring your enterprise application development as recommended in *Chapter 3*, *Getting Organized*, you will be keeping all of your entity objects in a Common Model workspace, using one task flow workspace per use case in your requirements, and collecting all of this in one **master workspace**. Because you only want to define your application roles once in the application, you apply security in the master workspace to cover the whole application.

Implementing ADF security

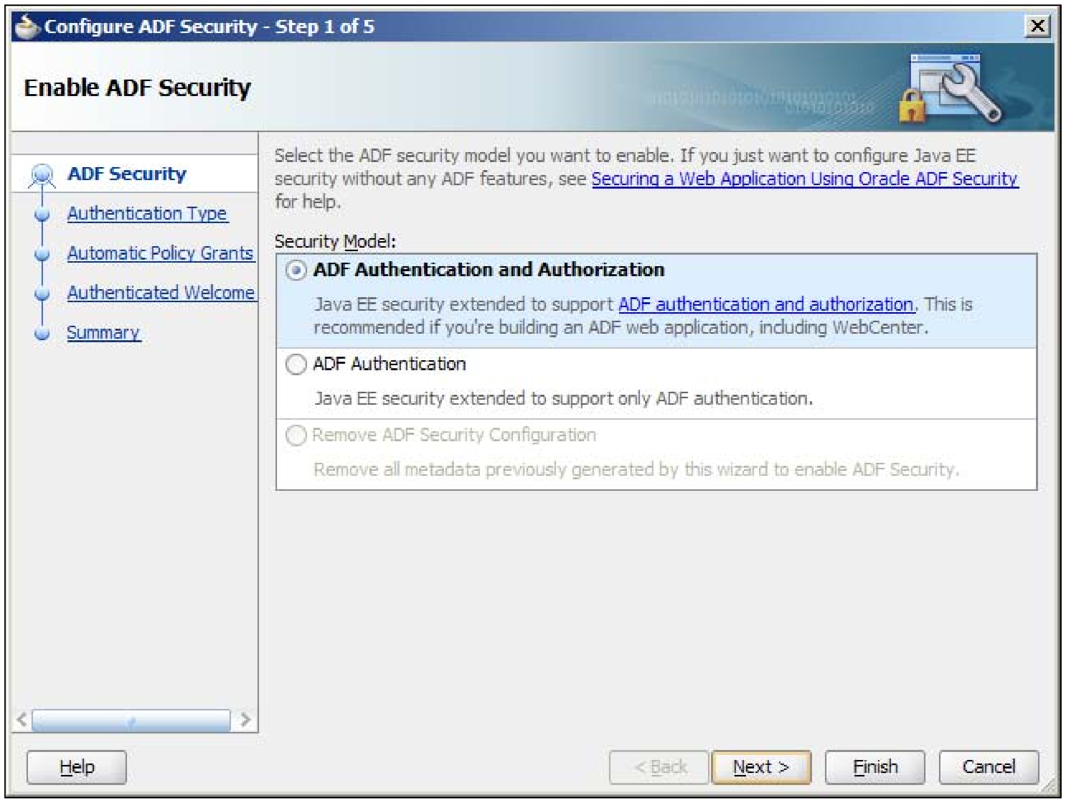
If you were to set up ADF security by hand, you would be editing a half-dozen complex XML files with complex interdependencies. Fortunately, JDeveloper offers to do all the hard work for you through the **Configure ADF Security** wizard.

To secure your ADF application, simply choose **Application | Secure | Configure ADF Security**. This wizard will take you through four steps to secure your application:

- Selecting a security model
- Selecting an authentication type
- Selecting how to grant access
- Select any common welcome page

Security model

In the first step of the wizard, you are asked to select a security model:



Select the ADF security model you want to enable. If you just want to configure Java EE security without any ADF features, see Securing a Web Application Using Oracle ADF Security for help.

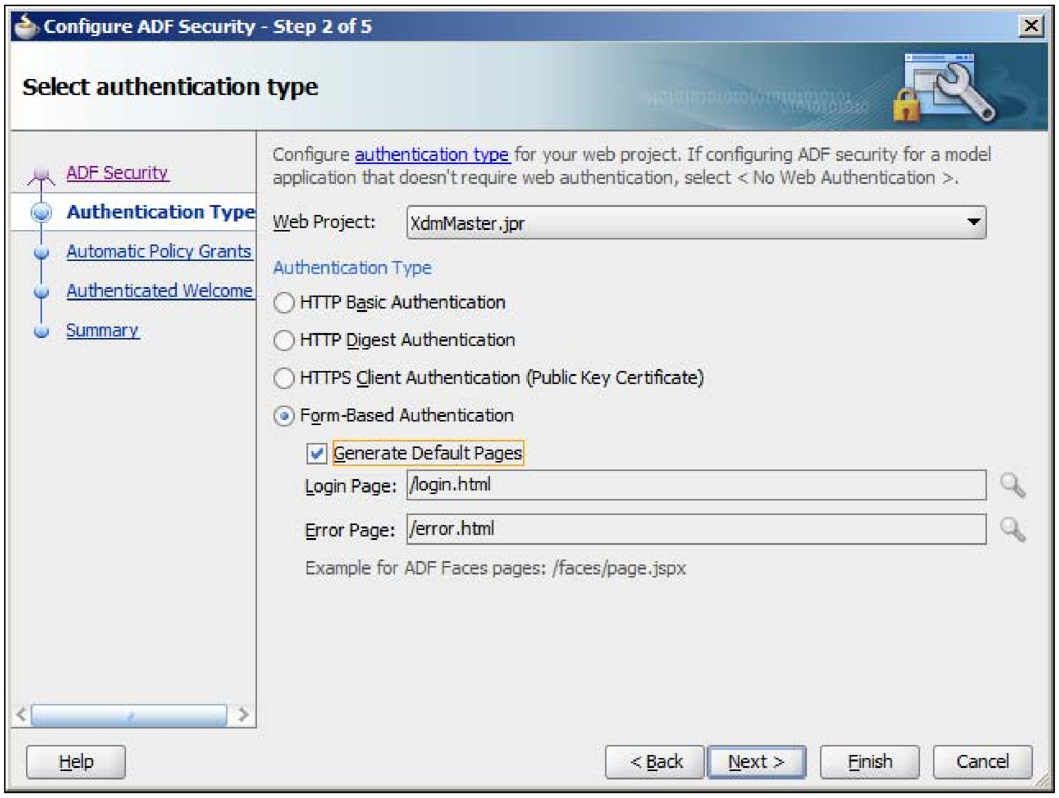
Security Model:

- [✓ ADF Authentication and Authorization] Java EE security extended to support ADF authentication and authorization. This is recommended if you're building an ADF web application, including WebCenter.
- ADF Authentication: Java EE security extended to support only ADF authentication.
- Remove ADF Security Configuration: Remove all metadata previously generated by this wizard to enable ADF Security.

Normally, you will always select **ADF Authentication and Authorization**. Even if your application does not distinguish between authenticated users, you still want to use ADF authorization to define which parts are accessible to anonymous users and which part to authenticated users.

Authentication type

Next, you are asked to choose the authentication type. You have several options, but as described previously, you would normally select **Form-based Authentication** in order to be able to control the login form displayed to the user:



Configure authentication type for your web project. If configuring ADF security for a model application that doesn't require web authentication, select < No Web Authentication >.

Web Project: XdmMaster.jpr

Authentication Type

( ) HTTP Basic Authentication
( ) HTTP Digest Authentication
( ) HTTPS Client Authentication (Public Key Certificate)
(●) Form-Based Authentication

[✓] Generate Default Pages

Login Page: /login.html

Error Page: /error.html

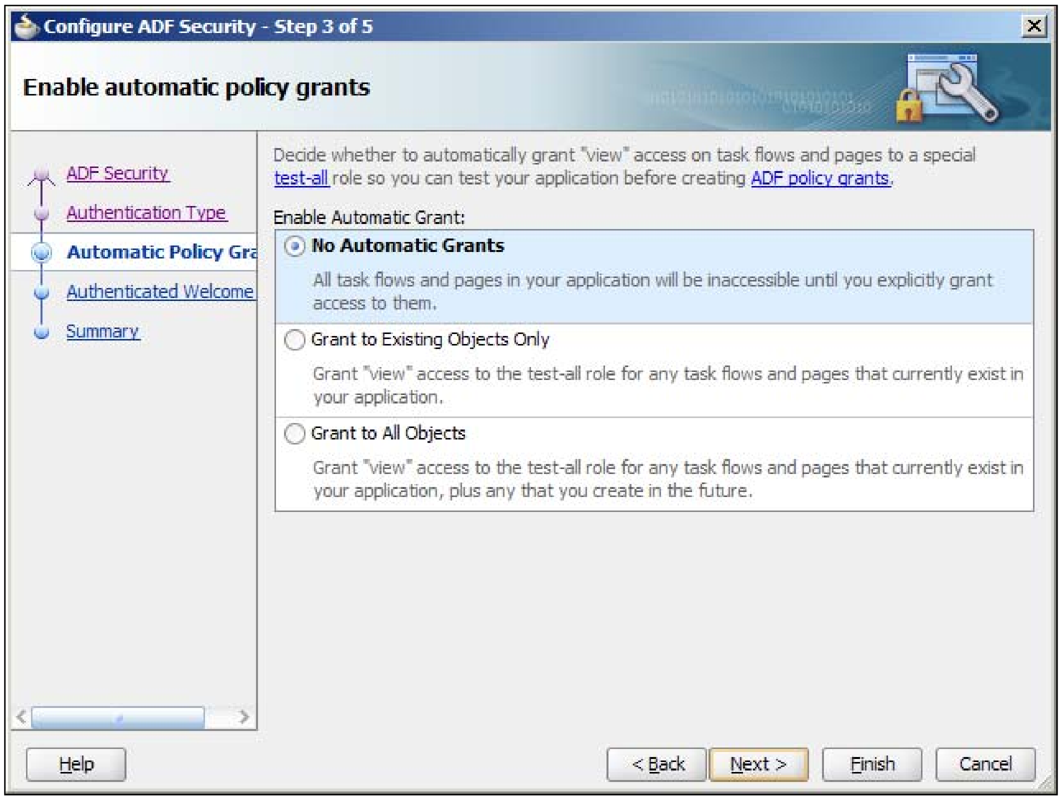
Example for ADF Faces pages: /faces/page.jspx

To demonstrate how form-based login works, you can ask JDeveloper to produce a default login page (and an error page if the authentication fails). In a real-life enterprise application, you would not use these default HTML pages, but instead build `.jspx` login and error pages as part of your application. Once you have examined the simple HTML pages and read the help topics about using your own `.jspx` pages as login and error pages, you can build the proper `.jspx` login and error pages and run the wizard again to set your application to use these pages.

For the purposes of the example in this chapter, select **Generate Default Pages**.

Access grants

In the third step of the wizard, you can decide how much access you want to grant to your task flows and pages. By default, securing your ADF application will secure all task flows and pages. This means that after you are done with the ADF Security wizard, you do not have access to any pages until you create test users and roles:



Decide whether to automatically grant "view" access on task flows and pages to a special test-all role so you can test your application before creating ADF policy grants.

Enable Automatic Grant:

[✓ No Automatic Grants]
All task flows and pages in your application will be inaccessible until you explicitly grant access to them.

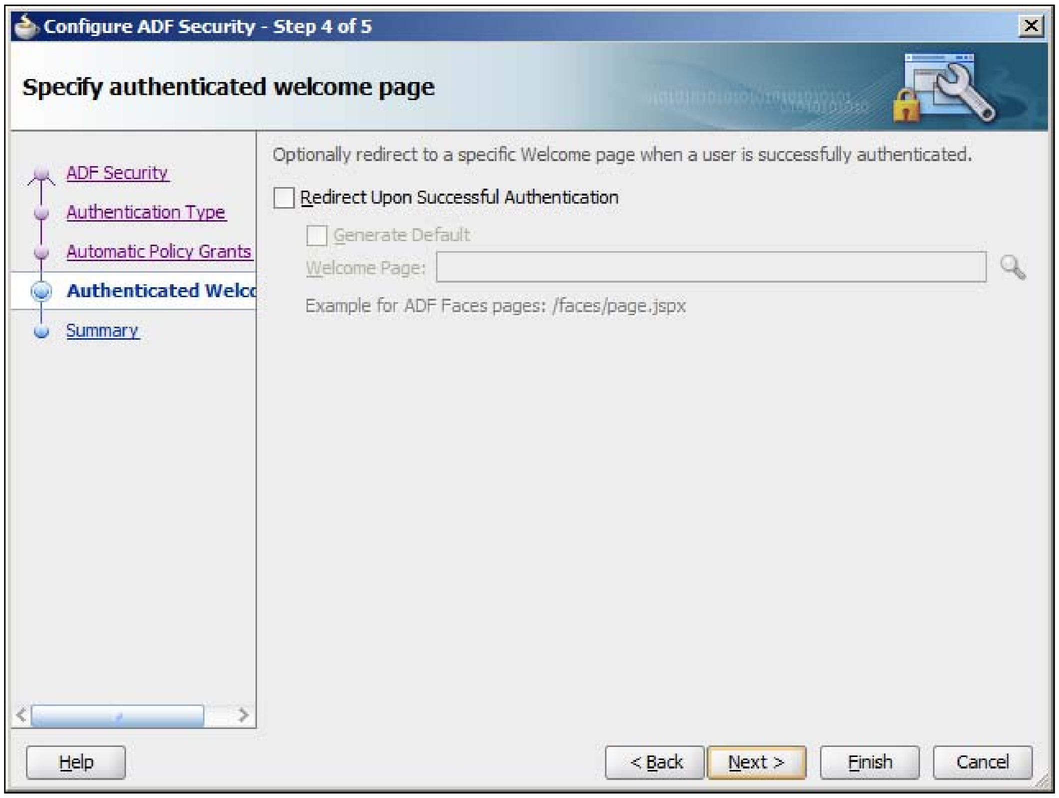
Grant to Existing Objects Only
Grant "view" access to the test-all role for any task flows and pages that currently exist in your application.

Grant to All Objects
Grant "view" access to the test-all role for any task flows and pages that currently exist in your application, plus any that you create in the future.

The most secure option is **No Automatic Grants**, which means that you will have to explicitly grant access to all task flows and pages. The other two alternatives will grant access to a special `test-all` role, either for all existing task flows and pages or for all existing and future task flows and pages. These two options are useful if you add security late in the project when your testers are already testing the application—by granting your testers the `test-all` role, you are not locking them out while you configure security for your application. If you choose this approach, be sure to check that you have removed grants to the test-all role before your application goes into production—there is a checkbox **Show … with test-all grants only** in the security dialog to help you find these grants.

Welcome page

The last decision you need to make is whether you will automatically direct your users to a common welcome page after successful login:



Optionally redirect to a specific Welcome page when a user is successfully authenticated.

☐ Redirect Upon Successful Authentication

☐ Generate Default

Welcome Page:

Example for ADF Faces pages: /faces/page.jspx

This matters because the ADF security framework will automatically intercept attempts to access a protected page and send the user to a login page. If you do not check this checkbox, your user will go back to the page he requested after successful login. If you do check this checkbox, your user will be sent to the common welcome page after login.

If you decide to redirect to a common start page, you should use the looking glass icon to select an existing page in your application.

When you click **Next** on this page, you are shown a summary of all the security XML files that will be changed. You do not have to worry about these, because JDeveloper will configure them all correctly based on your choices in the wizard.

Application roles

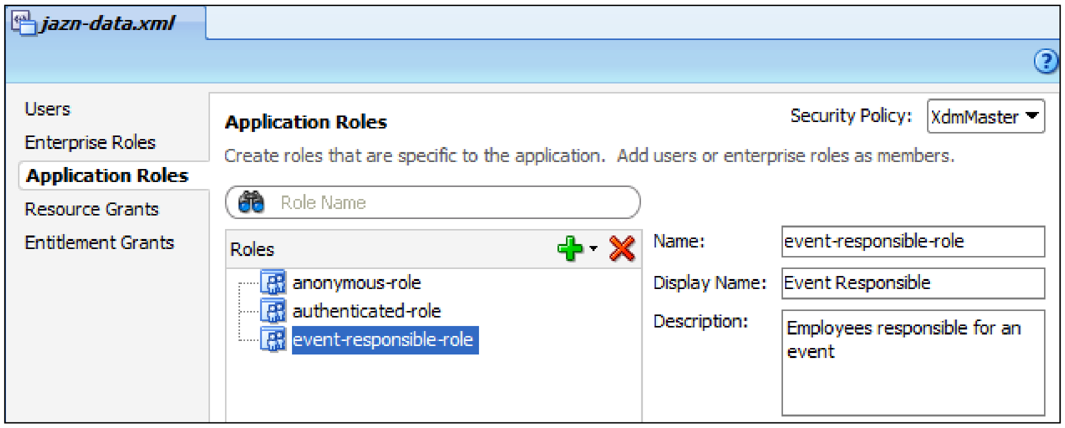
You, the application developer, are the person who understands the requirements and the pages, task flows, and entity objects that will be needed in the application. Therefore, you must determine the different roles that will be using the application.

The whole XDM application will make use of five roles:

- Admin Staff
- Operations Staff
- Finance
- Event Responsible
- Manager

As we have not built the whole application in this book, we will only define Event Responsible and Operations Staff here. Choose **Application** | **Secure** | **Application Roles** to open the application security configuration, stored in the `jazn-data.xml` file. The JDeveloper recognizes this file as part of the security configuration, and presents you with a nice overview tab—but you can click on the **Source** tab to see the raw XML data if you want to.

Click on the green plus sign next to **Roles** to create a new role, and fill in the **Name**, **Display Name**, and **Description** to the right:



Application Roles

Security Policy: XdmMaster

Create roles that are specific to the application. Add users or enterprise roles as members.

Name: event-responsible-role

Display Name: Event Responsible

Description: Employees responsible for an event

Repeat to create the **operations-staff** role with **Display Name Operations Staff**.

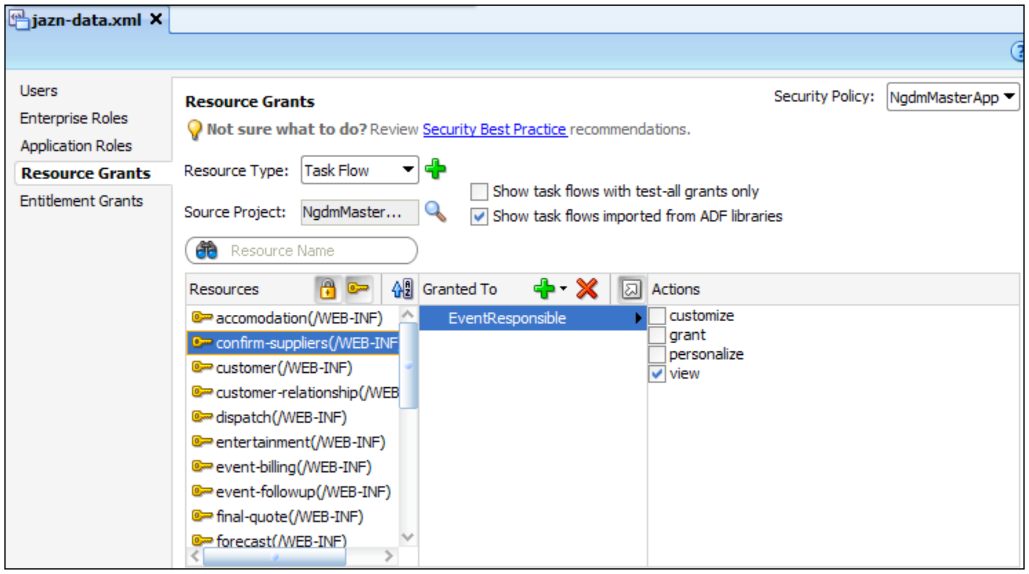
Implementing user interface security

In the user interface, you can apply security to either pages or task flows. Remember that the ADF Security wizard by default locks down everything, so you will not have access to any part of your application until you have explicitly granted access.

To apply security, choose **Application** | **Secure** | **Resource Grants**. If you already have the `jazn-data.xml` file open, you can also just select the **Resource Grants** tab in the left-hand side. Make sure you check the checkbox **Show task flows imported from ADF libraries** in order to see the task flows defined in the task flow workspaces.

On this page, select the resource type (**Web Page** or **Task Flow**), choose the resource and click the green plus sign to grant access to an application role. While securing the user interface for a normal ADF application, the only element under **Actions** you need to select is **view**. The other options are relevant if you deploy your application as a portlet to Oracle WebCenter, where it can be customized and personalized in different ways.

For the example in this chapter, assign the **person-timeline-flow** to the **event-responsible-role** and assign **task-overview-edit-flow** to both **event-responsible-role and operations-staff-role**.



Security Policy: NgdmMasterApp
Resource Type: Task Flow
Source Project: NgdmMaster...
[ ] Show task flows with test-all grants only
[✓] Show task flows imported from ADF libraries

Resources:
accomodation(/WEB-INF)
confirm-suppliers(/WEB-INF
customer(/WEB-INF)
customer-relationship(/WEB
dispatch(/WEB-INF)
entertainment(/WEB-INF)
event-billing(/WEB-INF)
event-followup(/WEB-INF)
final-quote(/WEB-INF)
forecast(/WEB-INF)

Granted To: EventResponsible

Actions:
[ ] customize
[ ] grant
[ ] personalize
[✓] view

All of your task flows should be shown with a key icon as shown previously—this indicates that access has been granted to at least one application role. If any of your task flows have the padlock icon, this means that nobody has been granted the right to execute them—in effect, they are locked and inaccessible.

Remember that your task flows are normally implemented with page fragments, and placed inside a dynamic region on a page. You must grant view privilege on both the page (often simply to the built-in role `authenticated-users`) and then more specific grants on the individual task flows.

> **New and improved: entitlements**
>
> In JDeveloper 11.1.1.4, a new feature was added: **Entitlement Grants**. If you have many resources, you can create entitlements to group resources together and then grant the whole entitlement to a role in one operation.

Implementing data security

In addition to the user interface security, it is also possible to apply security rules at the data level—to the entity objects.

Applying data security is an additional security layer that you can use to protect especially important or sensitive data. Your page fragments should, of course, only display the information that each user is entitled to see, but if you add data security at the entity level, you have an additional layer of protection. In an enterprise application that might be changed by a maintenance programmer five years after the project was initially built, this helps ensure that someone does not accidentally make data available to users who should not be able to view or change it.

Unfortunately, JDeveloper will only really help you implement data security if you place everything in the same workspace (which is not really a valid option for an enterprise application). There is no checkbox called **Show entity objects imported from ADF libraries**, so if you want to implement data security, you will have to do more work yourself.

> The following describes how to implement data security in an enterprise setting, where your entity objects are imported from an ADF library. If you keep everything in one workspace, there are simpler options. Refer to the online help in JDeveloper.

Implementing entity object security is a two-step process:

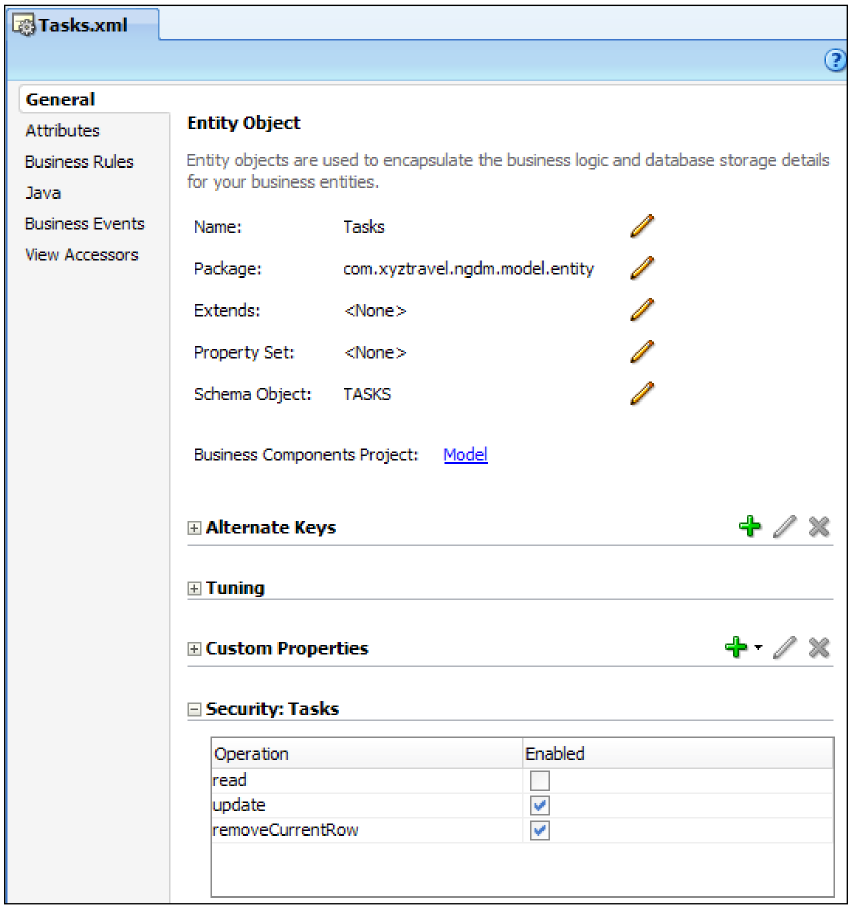
1. Define the operations you want to secure (read, update, delete)
2. Grant these operations to specific application roles

Defining protected operations

You define the operations you want to protect in the `CommonModel` project (or wherever you keep your entity objects). Data security is different from user interface security in that the data in an entity object is by default accessible, unless you decide otherwise. Only if you select to apply read, update, or delete security to an entity object is any checking performed.

Protecting an entity object

To protect a whole entity object (all attributes), you open it in JDeveloper, choose the **General** sub-tab and scroll down to the **Security** section. Here, you define which operations you want to secure. The operations that you *do not* select here are *not* secured, that is, if you leave **read** unchecked, all users can read the data presented by this entity object (if they have access to a task flow that uses it, of course):



Entity Object

Entity objects are used to encapsulate the business logic and database storage details for your business entities.

Name:	Tasks
Package:	com.xyztravel.ngdm.model.entity
Extends:	<None>
Property Set:	<None>
Schema Object:	TASKS

Business Components Project: Model

Operation	Enabled
read	☐
update	✓
removeCurrentRow	✓

Protecting an attribute

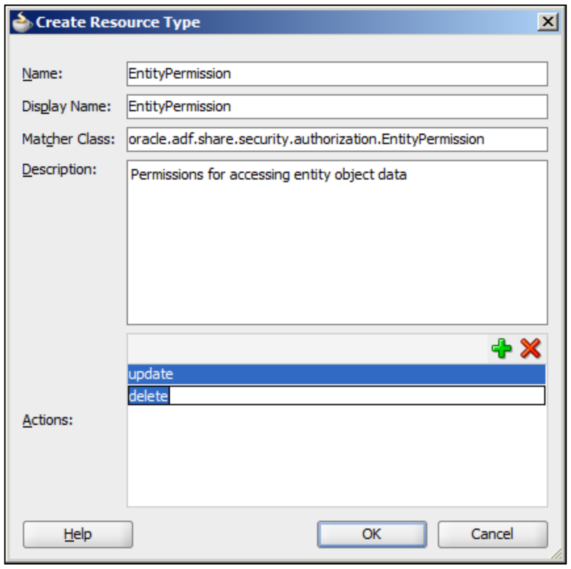
In addition to protecting a whole entity object, you can also protect the update operation for an individual attribute. To do this, go to the **Attributes** sub-tab and choose the attribute you want to protect. Scroll down to the Security section, open it and check the **update** checkbox.

Securing attributes secures the data, but does not change the user interface. If you remove the update privilege from an entity object, a user will not be able to change the value, but the user interface element does not automatically get disabled. For a good user experience, you should set the **ReadOnly** attribute with expression language so it is displayed as read-only if the user cannot change it. You can access user authorization in expression language using `#{securityContext…}` – this is documented in the *Fusion Developer's Guide for Oracle Application Development Framework* in the chapter on ADF Security.

Granting operations to roles

Now that you have defined which entity objects to protect, you can grant specific operations to specific application roles. This is done in the master application.

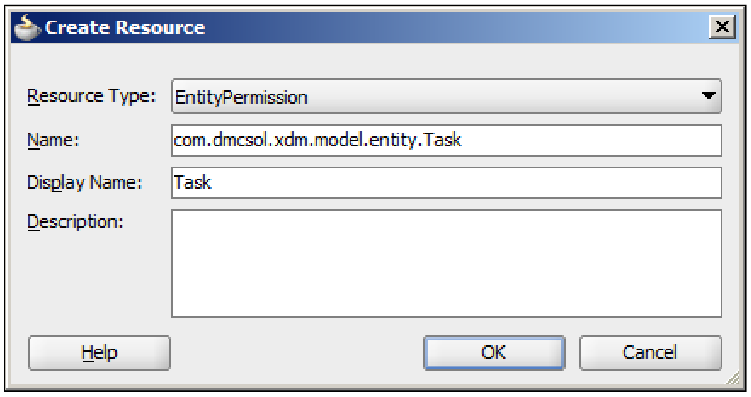
Go to the **Resource Grants** sub-tab in the `jazn-data.xml` dialog by choosing **Application** | **Secure** | **Resource Grants**. Your **Resource Type** drop-down will only contain **Task Flow** and **Web Page**, but you need to add another resource: an entity object. Click the green plus sign next to the **Resource Type** drop-down to bring up the **Create Resource Type** dialog:



Name:	EntityPermission
Display Name:	EntityPermission
Matcher Class:	oracle.adf.share.security.authorization.EntityPermission
Description:	Permissions for accessing entity object data
Actions:	update delete

Set the **Name** and **Display Name** fields to **EntityPermission** and set **Matcher Class** to **oracle.adf.share.security.authorization.EntityPermission**. Click the green plus sign to define the actions **update** and **delete** and click **OK**. If you wish the ability to protect read access as well, also add **read**. You only need to create one resource type for the entity objects with these two or three actions—you do not have to define each possible combination.

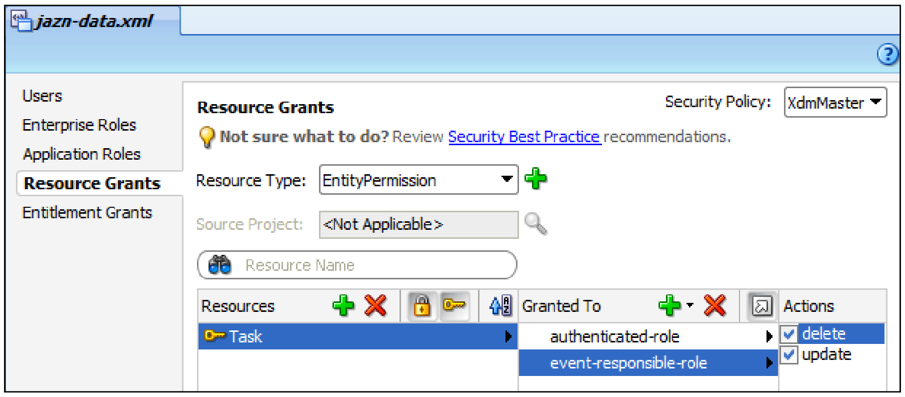
Select your new **EntityPermission** resource type and click the green plus sign under **Resources** to bring up the **Create Resource** dialog.:



Resource Type: EntityPermission
Name: com.dmcsol.xdm.model.entity.Task
Display Name: Task
Description:

In the **Create Resource** dialog, fill in the **Name** field with the full name of the entity object you wish to grant operations on, for example, **com.dmcsol.xdm.model.entity.Task**. Then click **OK**.

Now your resource should appear in the left-hand column. As for web pages and task flows, you can now click the green plus next to **Granted To** to select application roles and choose the operations allowed on this entity object for this role in the **Actions** column:



Security Policy: XdmMaster

Not sure what to do? Review Security Best Practice recommendations.

Resource Type: EntityPermission

Source Project: <Not Applicable>

Resources	Granted To	Actions
Task	authenticated-role	✓ delete
	event-responsible-role	✓ update

Even more data security

If you require even more sophisticated data security, you can use **Virtual Private Database** (**VPD**), which is a feature of the Enterprise Edition Oracle database. This technology allows you to associate policies with individual database tables and execute advanced PL/SQL to calculate additional restrictions on data access at runtime. In order to do this, you perform some VPD setup in the database and then override the `prepareSession()` method on the application module to send the actual logged-in user to the database before any data operations are executed.

Google *oracle adf vpd* for more information.

Users and groups

As a developer, you can define the application roles, but you do not know which users and groups are available in the organization that will be running the completed application. Therefore, your application roles must be **mapped** to the groups of users defined in the organization.

Enterprise roles or groups?

Same thing. Some identity management systems use the terminology that users are members of groups and others use the terminology that users are assigned enterprise roles. Even JDeveloper uses both terms—in version 11.1.1.4, the menu item is called **Groups**, but the tab it opens is called **Enterprise Roles**.

Mapping the application to the organization

Integrating your application server with your identity management system (Microsoft Active Directory, Oracle Internet Directory, or some other system) is the task of your application server administrator. This procedure is outside the scope of this book, but is well documented in the Oracle Fusion Middleware documentation—start with the *Oracle Fusion Middleware Security Overview* manual and follow the references it provides.

When this task is done, the application server knows the enterprise roles (or groups) that are used in your organization. This allows the person deploying your ADF application to the application server to perform the mapping of application roles (defined by the application developer) to enterprise roles (defined by the organization).

This approach means that the application developer is free to define the application roles that make sense in the application, and the organization is free to define groups that correspond to their organization.

Example users and enterprise roles

You do not normally integrate the built-in WebLogic server that comes with JDeveloper with your identity management system, but you will still need to test the security of your application. You normally do this by using a simple file-based identity management system that you can set up using dialog boxes in JDeveloper. Because the users you set up here only exist in your own system, you are free to create and delete users as needed.

With the simple, local solution, you can:

- Define test users
- Define test enterprise roles
- Assign members to your enterprise roles
- Assign application roles to enterprise roles

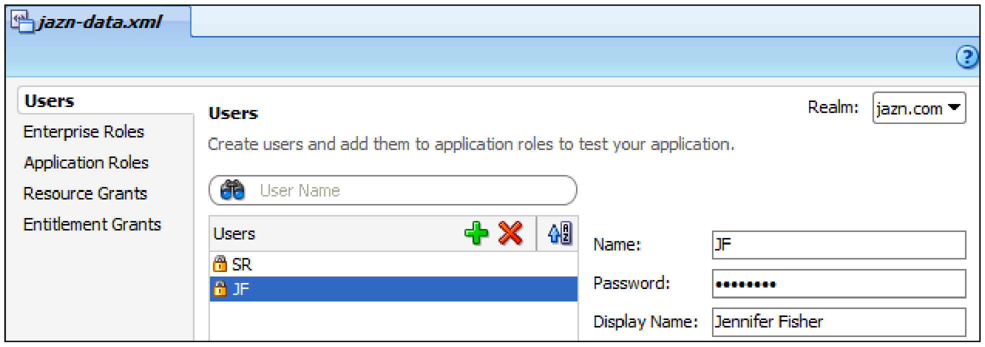
The first three steps establish an identity infrastructure like you might have in Microsoft Active Directory or Oracle Internet Directory in your production environment, and the last step mimics the task the application deployer performs when putting your application in production.

To create your example users, you choose **Application** | **Secure** | **Users** to bring up the `jazn-data.xml` file. This file represents a simple user repository and can be used for testing your security. JDeveloper recognizes this file as part of the security configuration, and presents you with a nice overview tab—but you can click on the **Source** tab to see the raw XML data if you want to. Click on the green plus sign to add a user and provide a name and a password.

Password policy

The password must satisfy the security policy of the built-in WebLogic application server—in JDeveloper 11.1.1.4, which was current at the time of writing, the requirement is simply that the password must be at least eight characters. Some versions require both letters and numbers.

For the purposes of this chapter, create users *SR* (*Steven Robertson*) and *JF* (*Jennifer Fisher*):



Users

Create users and add them to application roles to test your application.

Realm: jazn.com

SR

JF

Name: JF

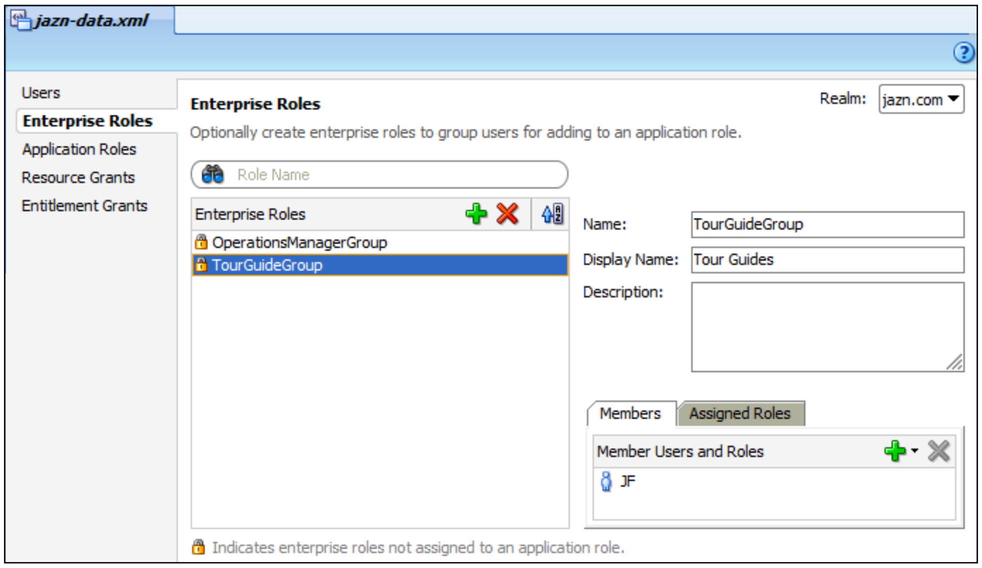
Password: ••••••••

Display Name: Jennifer Fisher

Note the little padlock icon next to these users—it indicates that they do not have access to anything yet.

Then change to the **Enterprise Roles** tab (or choose **Application** | **Secure** | **Groups**) to define the groups that your users will be members of. If you are building an application for in-house use where you already know the groups that the organization uses, you should define all or some of the real user groups. If you are working for an independent software vendor such as DMC Solutions, and you do not know the organization where your application will be deployed, create some example user groups.

For the XDM application, create the enterprise roles **OperationsManagerGroup** and **TourGuideGroup**. In the **Members** sub-tab, add **SR** as member of operations managers and **JF** as member of tour guides:



Enterprise Roles

Optionally create enterprise roles to group users for adding to an application role.

Realm: jazn.com

OperationsManagerGroup

TourGuideGroup

Name: TourGuideGroup

Display Name: Tour Guides

Description:

JF

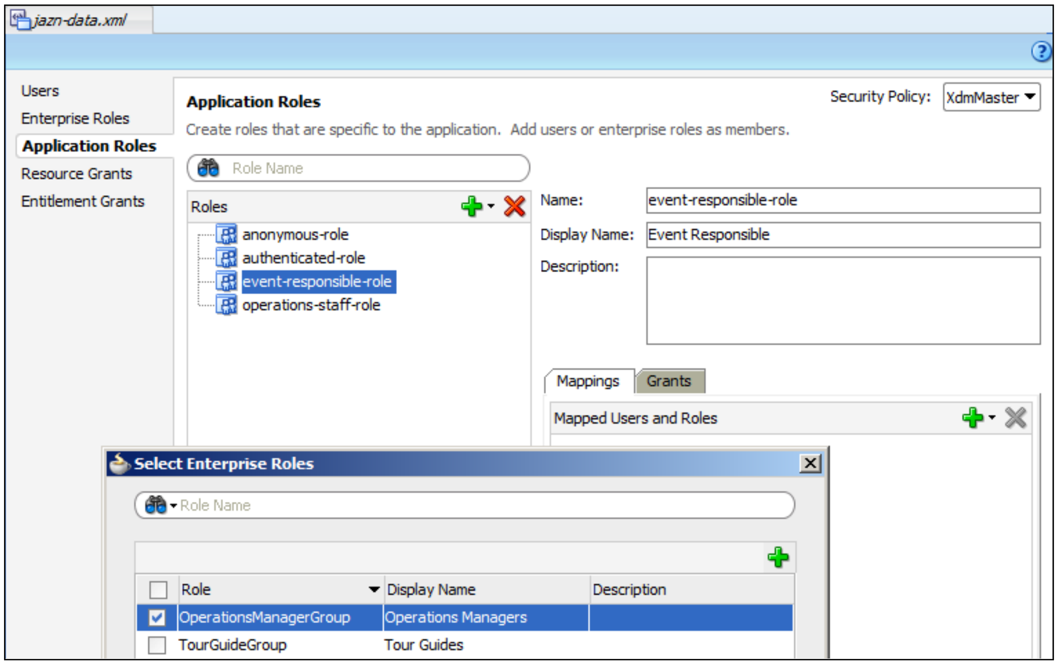
Indicates enterprise roles not assigned to an application role.

Again, note the padlock icon indicating that these enterprise roles do not have access to anything yet—they have not been assigned any application roles.

Assigning application roles

To assign the application roles you have defined to your enterprise roles/user groups, switch back to the **Application Roles** sub-tab where you defined the application roles. Here, you use the **Mappings** sub-tab to assign enterprise roles to each application role.

Select the **event-responsible-role** and click the green plus next to **Mapped Users and Roles**. Select **Add Enterprise Role** and check the checkbox for the enterprise role **OperationsManagerGroup**:



	Role	Display Name	Description
✓	OperationsManagerGroup	Operations Managers	
	TourGuideGroup	Tour Guides	

Similarly, map the enterprise role **TourGuideGroup** to the application role **operations-staff-role**.

Anonymous and authenticated

Note that you have two application roles available in addition to those you defined yourself: **anonymous-role** and **authenticated-role**. If you want some pages or task flows to be accessible to everyone without requiring a login, you can grant access to these pages or task flows to **anonymous-role**. If there are pages or task flows that you want every authenticated user to be able to see, irrespective of the enterprise role or group membership, grant access to these pages and task flows to **authenticated-role**.

Running the application

When you have saved your `jazn-data.xml` file, you can run the application as usual (by right-clicking on the **Xdm.jspx** page and choosing **Run**). You will be met with a very simple username/password page—this is the auto-generated `login.html` page that JDeveloper built for you.

If you log in with the user *SR*, you will see the application working as before, because user *SR* is a member of the Operations Manager group, which has the Event Responsible role assigned, which gives access to the whole application.

If you log in with user *JF*, on the other hand, you will see a blank page if you select the **Timeline** menu item. This is because *JF* is only a member of the Tour Guide group, which has the Operations Staff application role, which does not give access to the person timeline task flow.

Removing inaccessible items

For the best user experience, users should not be able to select menu items that will not show them anything. You can handle this in two ways:

- You can hide the menu item completely by setting the **Rendered** attribute to **false**. Use this if you do not want to confuse your users with menu items that are not relevant to them
- You can show the menu item disabled (grayed out) by setting the **Disabled** attribute to **true**. Use this if you want to show your users that additional functionality exists but is not accessible to them

You can use expression language to set attributes on menu items that are not accessible to a user. As mentioned previously, you have access to the current security context in Expression Language using `#{securityContext…}` functions.

To disable the **Timeline** menu item for users who do not have access to the timeline task flow, you can set **Disabled** to **!#{securityContext.taskflowViewable['/WEB-INF/person-timeline-flow.xml#person-timeline-flow']}**.

Refer to the ADF Security chapter in *Fusion Developer's Guide for Oracle Application Development Framework* for details on accessing the security context using expression language.

Summary

In this chapter, you have applied ADF security to your master application so that users would be prompted to log in when starting the application. We just used the default login page, but you can easily build your own login page and integrate this with the application. You defined application roles and implemented security on pages, task flows, and entity objects, specifying which application roles can do what. Finally, you used the built-in user and group repository to test your application.

You can tell your manager that the XDM application is secure and implements all required logic to ensure that only properly authenticated and authorized users can access functionality and data. He is happy to hear that and asks you to package the application for deployment to the test server. That is the subject of the next (and last) chapter.

11
Package and Deliver

When a tailor has taken all the measurements of his customer, he cuts pieces of cloth according to the agreed style. However, 36 pieces of cloth do not make a jacket—just like 36 individual ADF Libraries do not make up an application.

What remains is to sew all the parts together into one deliverable package for installation—first on your test environment and then on your production environment. This package should include your executable code, any database scripts, and the necessary documentation.

After each cycle of test and rework, you need to create a new deliverable package until it passes all the tests. Then the *same* package goes to the operations staff to install in the pre-production or production environment. In case your package does not install cleanly on this environment, you go back to the drawing board, fix the code or the documentation and create a new package.

This deployment happens from the master application that includes all the common ADF Libraries as well as all the task flow libraries containing your business components, web pages, and task flows.

What is in the package?

When the development team is done and hands the application over to production, the package should include:

- The runnable application
- Any database code
- Installation and operation instructions

The runnable application

The enterprise ADF application you and your team have built is a **Java Enterprise Edition** (**JEE**) application. Therefore, it is delivered in the standard JEE application form as a Java Enterprise Archive (`.ear`) file.

An `.ear` file is just a compressed file containing application code and a bit of metadata. You can open the file with an unzip utility to see what is inside, normally, the `.ear` file contains a Web Archive (`.war`) file that you can again unzip to see inside.

Your EAR file will contain the application roles and other security features you configured in *Chapter 10*, *Securing your ADF Application*.

Database code

If your application contains new or changed database objects, you will, of course, need to supply SQL scripts for your database administrator to run. If you are using advanced security such as Virtual Private Database, there will also be database scripts to run to implement this. As this is the same for all database applications, we will not discuss SQL scripts much in this chapter.

Installation and operation instructions

Finally, your installation package needs to include the necessary instructions for your database and application server administrators to install the application.

Your instructions must include exactly which version of WebLogic you need—there is a WebLogic version matching each version of JDeveloper. You also need to point your application server administrator to the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework* for instructions on how to prepare the WebLogic environment for ADF.

In addition to the basic instructions ("deploy this file", "run this script"), your installation instructions must include the name of any database connections the application uses, so that the application server administrator can make sure this connection name is available. You will also have to provide the name and intended usage for each of the application roles your application defines, so the application server administrator or security administrator can map these roles to the user groups or enterprise roles already defined in the organization.

Finally, you need to tell your system administrators where your system writes its log files. If you are using ADFLogger (which is recommended), instruct your system administrators how to configure the `logging.xml` file.

Preparing for deployment

Preparing your application for deployment involves cleaning up the code and setting application parameters for production use.

Cleaning up your code

Just as the tailor should not leave pins in your finished jacket, you should not leave development artifacts in your installation package. Some things that might accidentally slip into the deployment package include:

- Database connections
- Test users and groups
- Print statements
- Debug settings in `web.xml`

Additionally, JDeveloper contains a code audit tool. To see what JDeveloper thinks about your code, select a project and choose **Build | Audit**. In the **Audit** dialog box, you can click **Edit** to select which rules you want to check in your project.

Database connections

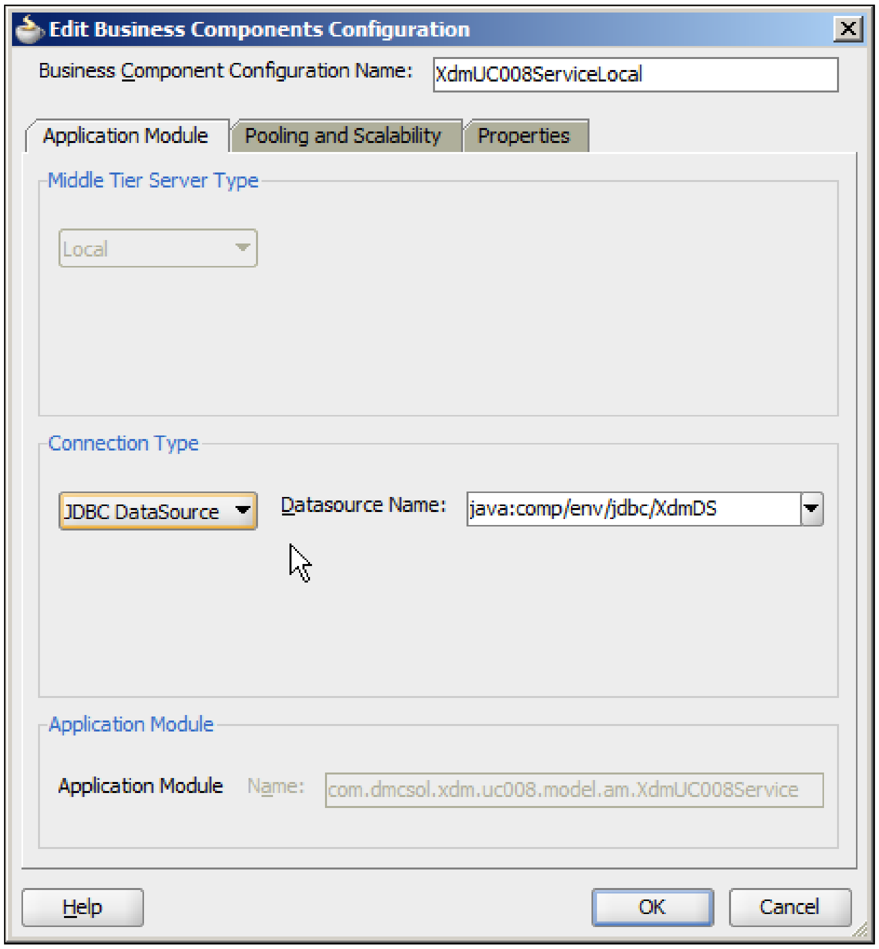
Remember that when you created your entity objects, you also created a database connection. By default, this connection includes the server, port, and database name. That is OK while you are running your code against a development database, but of course, not OK when you are done and want to deploy to the test or production server.

The solution is to use a named **DataSource** instead. With DataSources, your application just contains the name of a DataSource, and it is up to the application server administrator to create a DataSource with that name on the application server. This approach gives the server administrator the freedom he needs to administrate the production environment, and even move the application and database around between servers if necessary.

Using DataSources from the beginning

As described in *Chapter 6, Building the Enterprise Application*, you should set your application modules to a DataSource instead of a JDBC URL as soon as your create the application module.

If you are not certain that you have used DataSources everywhere, you can right-click on each application module and choose **Configurations**. The **Manage Configurations** dialog appears. Select the configuration that ends with ...`Local` and click **Edit** to open the **Edit Business Components Configuration** dialog. Under **Connection Type**, check that you have selected **JDBC DataSource** (and *not* **JDBC URL**) and make a note of the DataSource name for your installation documentation:


Edit Business Components Configuration

Business Component Configuration Name: XdmUC008ServiceLocal

Middle Tier Server Type: Local

Connection Type: JDBC DataSource

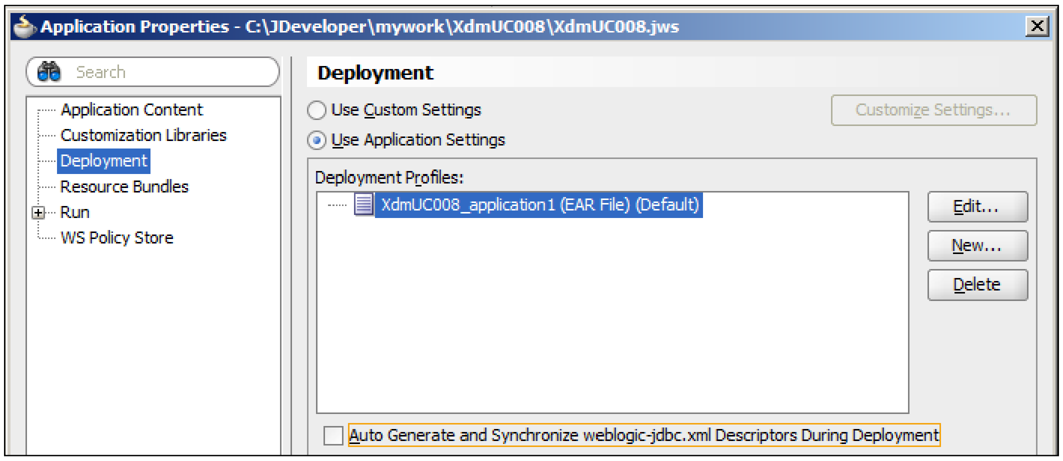
Datasource Name: java:comp/env/jdbc/XdmDS

Application Module

Application Module Name: com.dmcsol.xdm.uc008.model.am.XdmUC008Service

The reason that you can use either a URL or a DataSource while developing is that by default, JDeveloper will automatically create a DataSource for you when you deploy your application to the built-in server or an external WebLogic server. That is not what you want when delivering your application, so you need to change this setting.

This is done in the application deployment profile that you can access by choosing **Application | Application Properties** to bring up the **Application Properties** dialog and then choose **Deployment**:



Use Custom Settings [✓ Use Application Settings]

Deployment Profiles:

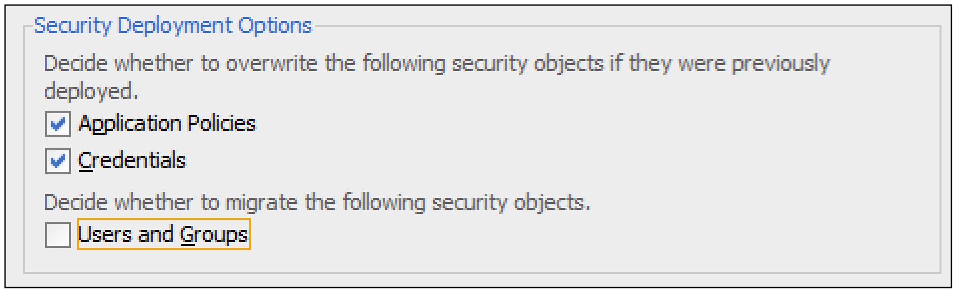
XdmUC008_application1 (EAR File) (Default)

☐ Auto Generate and Synchronize weblogic-jdbc.xml Descriptors During Deployment

Deselect the checkbox **Auto Generate and Synchronize weblogic-jdbc. xml Descriptors During Deployment** in order to ask the JDeveloper to stop automatically creating DataSources for you.

Test users and groups

You might remember from *Chapter 10, Securing your ADF Application* that we created test users and groups in order to test our security settings. A WebLogic server running in development mode (such as your stand-alone development server) will accept any users and groups deployed as part of the application, so you need to tell the JDeveloper not to package these into your final application. To do this, choose **Application | Application Properties** to bring up the **Application Properties** dialog and then choose **Deployment**:



Decide whether to overwrite the following security objects if they were previously deployed.

✓ Application Policies

✓ Credentials

Decide whether to migrate the following security objects.

☐ Users and Groups

Under **Security Deployment Options**, deselect the **Users and Groups** checkbox. Then click **OK** to close the dialog.

Depending on the choices you made in the ADF security wizard, you might also have a `test-all` role with access to all screens and task flows in your application. If you allowed a `test-all` role, you should use the security dialogs as described in *Chapter 10*, *Securing your ADF Application*, to make sure that your final application does not contain any grants to the test-all role.

Other development artifacts

Of course, you have used the logging method you all agreed on in the project team and did not write any simple `System.out.println()` statements in your code. However, somebody else might have done so. To check for this kind of impurities in your project code, you can use JDeveloper's global file search capability. Choose **Search | Find in Files** to search through your active project or application (or any user-defined path in the file system).

In *Chapter 8*, *Look and Feel*, we set `org.apache.myfaces.trinidad.DISABLE_CONTENT_COMPRESSION` in the `web.xml` file to `true` in order to see full names of CSS styles. This setting should be set back to `false` before you deploy your application to production. If you changed other settings for development or debug purposes, you should also set these back to the default.

Setting the application parameters for production use

In addition to making sure you do not have development and debugging stuff in your application, you want to make sure that your application has the right stuff—the configuration settings necessary for good performance.

Application module tuning

In the **Edit Business Components Configuration** dialog, you can also optimize the performance of your ADF application on the **Pooling and Scalability** tab.

The most important parameter on this tab is the **Referenced Pool Size** parameter. The ADF framework will keep this number of application module instances in memory (default **10**), ready to serve your users. As long as you have fewer concurrent users than this value, each user will get the same application module each time his browser communicates with the application server. Once you have more users than this number, users will have to share application modules. To allow this sharing, ADF has to store all of the internal state of the application module in the database in order to free it up for another user. This is called **passivation** and is an expensive operation; so as long as your application is running on one server, you want to avoid it as much as possible.

To minimize passivation, set **Referenced Pool Size** as close to the maximum number of concurrent users you expect as possible. Of course, if you set this value very high and run out of memory on your application server, your application server will start swapping memory to disk, which is even more expensive. You need to work with your application server administrator to find the *sweet spot* where application modules are not passivated too often, and your application server machine does not run out of memory.

There are many ADF tuning settings, and they can be set for each application module individually to match how that application module is expected to be used. The online help describes each setting, and the *Oracle Fusion Middleware Performance and Tuning Guide* gives tuning guidelines.

Controlling database locking

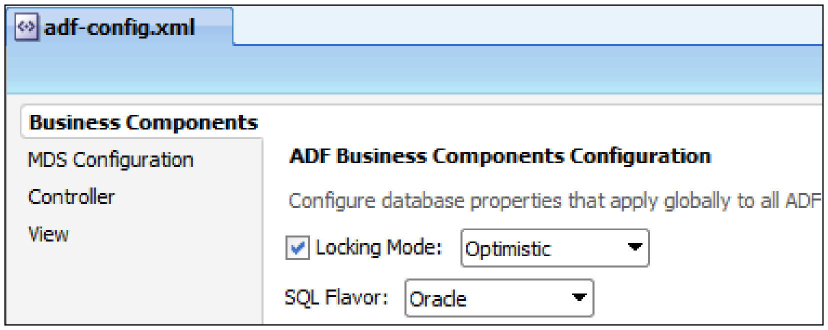
When several people are using a database application at the same time, they might attempt to change the same record in the database at the same time. This situation can be resolved in two ways:

- With pessimistic locking
- With optimistic locking

Pessimistic locking assumes that this conflict is likely (hence "pessimistic") and locks the record as soon as the first user tries to change it. This has the advantage that the second user will not be allowed to change the value at all. On the other hand, this has the disadvantage that if the first user goes away without changing the record, the lock remains and the second user is barred from changing the record. The server cannot tell if his browser crashed or if he has left for a three-week vacation, so it has to perform cleanup of locked records on some schedule. In an ADF web application, this happens when your web session times out.

Optimistic locking assumes that this conflict is not likely (hence "optimistic"). The first user is allowed to change his copy of the record, and the second user is allowed to change her copy. This has the advantage that the application server does not have to perform regular unlocking of stuck records, and nobody has to wait for these locks to be released. The disadvantage to this approach is that if the second user commits her change to the database first, the first user will be turned away—in effect, he will be told that his change cannot be stored in the database, because another user has already changed the value.

Oracle recommends the default setting **Optimistic** for web applications like the one we build in this book (web applications make up 95 percent or more of all ADF applications). You can change the global default in the **adf-config.xml** file that you find under **Application Resources**, **Descriptors**, **ADF META-INF**. On the **Business Components** sub-tab of the **Overview** tab, you can set the default locking mode for the whole application:



ADF Business Components Configuration

Configure database properties that apply globally to all ADF

✓ Locking Mode: Optimistic

SQL Flavor: Oracle

Tuning your ADF application

A whole book could be written on ADF tuning—but this book is not that book. There is a lot of good ADF tuning advice in the chapter on *Oracle Application Development Framework Performance Tuning* in *Oracle Fusion Middleware Performance and Tuning Guide*. Additionally, a lot of material on ADF tuning is available on the Oracle Technology Network and elsewhere on the internet—google *oracle adf tuning* for more.

Setting up the application server

While developing, you were probably just clicking **Run** to run your application. As you have noticed, this means that your application runs in the built-in WebLogic server on your local machine. This instance of WebLogic is pre-integrated with JDeveloper, so it contains all necessary libraries and settings for running the ADF applications.

In addition to these built-in WebLogic instances, your development server administrator must set up at least a test/integration server where you can collect all the components of the application for integration testing. Your environment might also include pre-production or other WebLogic server instances before the final production environment.

A minimum environment would use:

- The WebLogic server integrated with JDeveloper for development on each developer's workstation
- A stand-alone server for integration and test
- The production environment

In an enterprise setting, you should use:

- The WebLogic server integrated with JDeveloper for development on each developer's workstation
- A stand-alone development server for integrating the code
- A stand-alone test server that your test team will work on
- A pre-production server that your operations people can use to test the final deployment and where stress/performance tests on be run
- The production server
- Depending on how much training you plan to give your users, you might also need an education system

As there are licensing costs involved, your IT manager is also likely to have an opinion on how many environments you need.

Who is in charge?

You need one stand-alone server that the development team has full root/Administrator access to. During your project, there are always going to be times when you need to make some code available for a demo or a discussion of possible solutions without having to involve people and procedures outside the team.

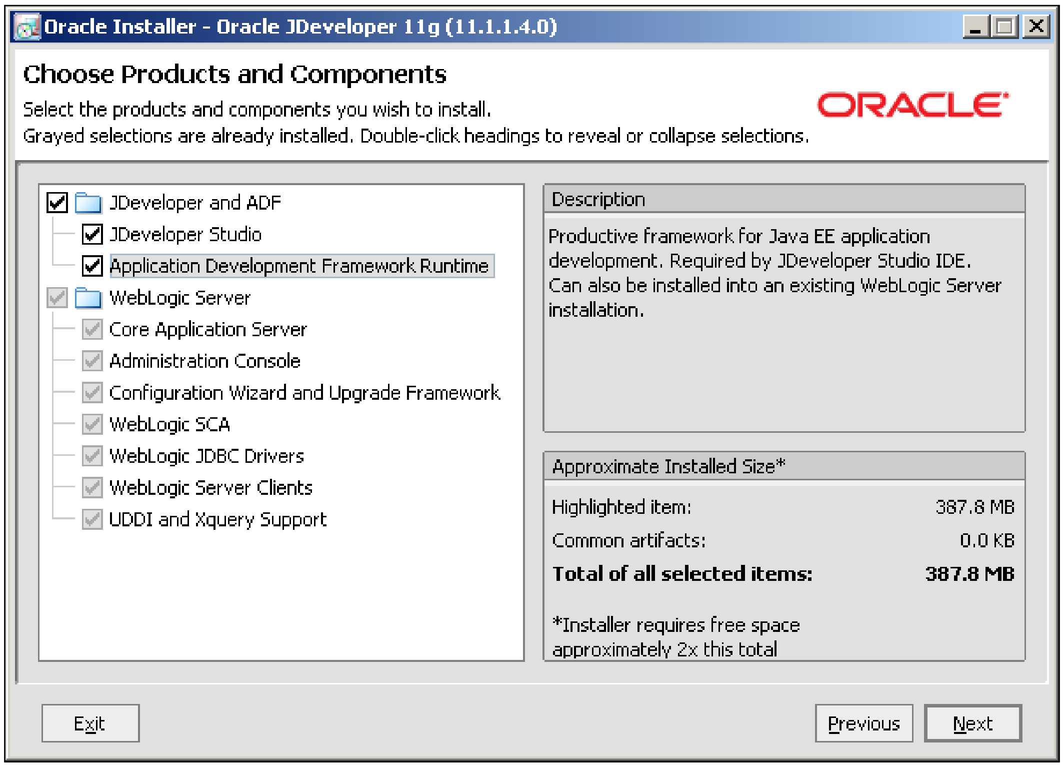
In an enterprise setting, the operations group manages all the other servers, following the systems management procedures and guidelines of your organization.

Installing the ADF Libraries

When your server administrator prepares one of your WebLogic servers, he will download the production software from `http://edelivery.oracle.com` and install the necessary ADF Libraries as documented in the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework.*

However, for the stand-alone server you manage in the development team, you can simply install a WebLogic server downloaded from the Oracle Technology Network and use the JDeveloper installer to install the ADF runtime libraries on it. To do this, copy the JDeveloper installer to the server where you have already installed WebLogic, and start it. When prompted to select a Middleware home directory, choose the directory where your existing WebLogic installation can be found.

In the next step of the wizard, choose to install **Application Development Framework Runtime**. You should also install **JDeveloper Studio** on the application server as well to get access to some additional tools and libraries that you will need if you decide to run automatic builds on your application server:


Oracle Installer - Oracle JDeveloper 11g (11.1.1.4.0)

Choose Products and Components

Select the products and components you wish to install.
Grayed selections are already installed. Double-click headings to reveal or collapse selections.

- [✓] JDeveloper and ADF
  - [✓] JDeveloper Studio
  - [✓] Application Development Framework Runtime
- [✓] WebLogic Server
  - [✓] Core Application Server
  - [✓] Administration Console
  - [✓] Configuration Wizard and Upgrade Framework
  - [✓] WebLogic SCA
  - [✓] WebLogic JDBC Drivers
  - [✓] WebLogic Server Clients
  - [✓] UDDI and Xquery Support

Description

Productive framework for Java EE application development. Required by JDeveloper Studio IDE. Can also be installed into an existing WebLogic Server installation.

Approximate Installed Size*

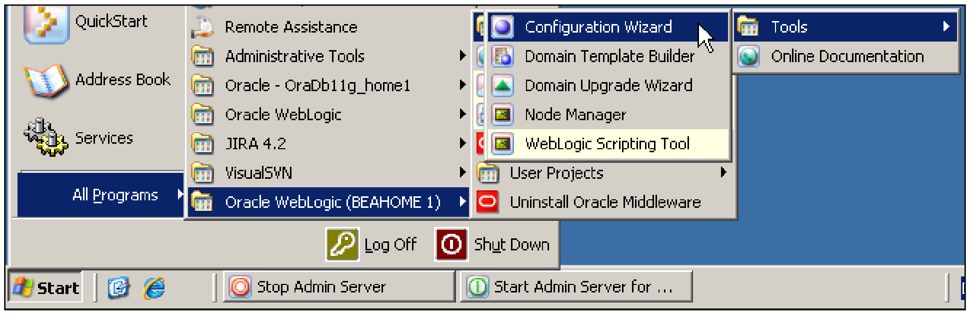
Highlighted item:	387.8 MB
Common artifacts:	0.0 KB
Total of all selected items:	**387.8 MB**

*Installer requires free space approximately 2x this total

Click **Next** through the rest of the installation to install the ADF libraries.

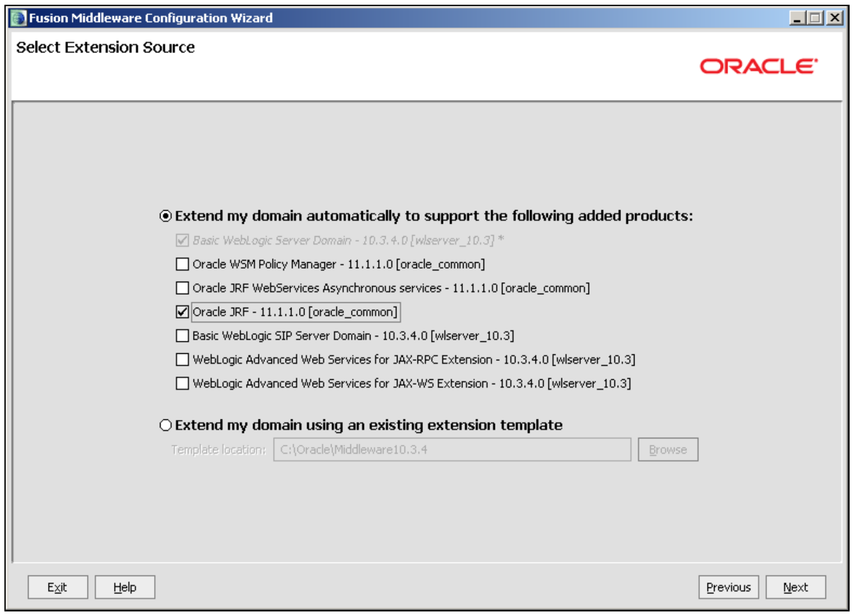
Setting up your domain

Your operations people will also set up the domain on their servers—but on your own server, you need to extend the existing WebLogic domain with the ADF Libraries yourself. This is done through the **Configuration Wizard**—on Windows, this can be found under **All Programs** with a path something like **Oracle WebLogic (BEAHOME 1)** | **WebLogic Server 11gR1** | **Tools** | **Configuration Wizard**:



In this wizard, first choose to extend an existing domain and then choose your WebLogic domain directory. In a default install on Windows, this can be found under `C:\oracle\Middleware\user_projects\domains`.

In the next step of the wizard, choose to extend the domain with Oracle JRF (Java Required Files):



Select Extension Source

[✓ Extend my domain automatically to support the following added products:]

- [✓] Basic WebLogic Server Domain - 10.3.4.0 [wlserver_10.3] *
- [ ] Oracle WSM Policy Manager - 11.1.1.0 [oracle_common]
- [ ] Oracle JRF WebServices Asynchronous services - 11.1.1.0 [oracle_common]
- [✓] Oracle JRF - 11.1.1.0 [oracle_common]
- [ ] Basic WebLogic SIP Server Domain - 10.3.4.0 [wlserver_10.3]
- [ ] WebLogic Advanced Web Services for JAX-RPC Extension - 10.3.4.0 [wlserver_10.3]
- [ ] WebLogic Advanced Web Services for JAX-WS Extension - 10.3.4.0 [wlserver_10.3]

Extend my domain using an existing extension template

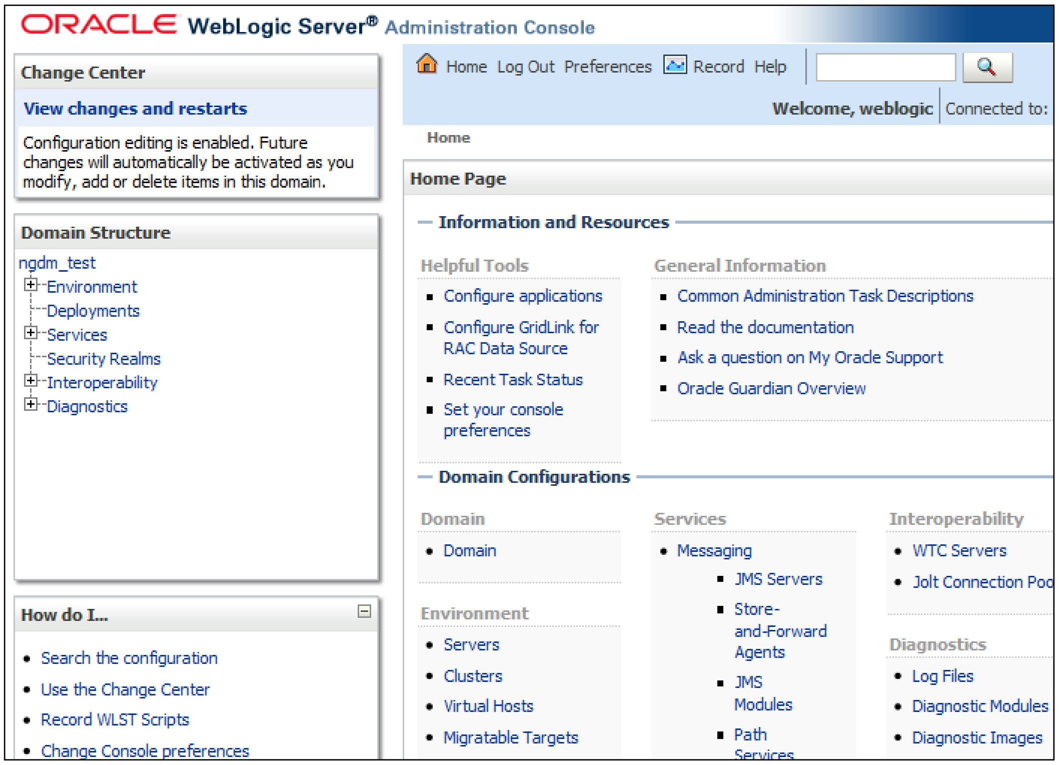
Template location: C:\Oracle\Middleware10.3.4

In following the steps in the wizard, you might be prompted to configure your JDBC data source—if so, give the name you used in your application development and point to the database you wish to run the test instance of your application against.

Creating a DataSource on the server

If the configuration tool did not prompt you to create a DataSource on the server, you can use the console (a web-based administration interface that comes with the Oracle WebLogic application server).

To start the console, open a web browser and enter the URL to your application installation, including the port, followed by `/console` (for example, `http://test.dmcsol.com:7001/console`). The WebLogic console then appears (it takes a bit of time for the application to load the first time you use the console):


ORACLE WebLogic Server® Administration Console

Home Log Out Preferences Record Help

Welcome, weblogic Connected to:

Home

Change Center

View changes and restarts

Configuration editing is enabled. Future changes will automatically be activated as you modify, add or delete items in this domain.

Domain Structure

ngdm_test
- Environment
- Deployments
- Services
- Security Realms
- Interoperability
- Diagnostics

How do I...

- Search the configuration
- Use the Change Center
- Record WLST Scripts
- Change Console preferences

Home Page

Information and Resources

Helpful Tools
- Configure applications
- Configure GridLink for RAC Data Source
- Recent Task Status
- Set your console preferences

General Information
- Common Administration Task Descriptions
- Read the documentation
- Ask a question on My Oracle Support
- Oracle Guardian Overview

Domain Configurations

Domain
- Domain

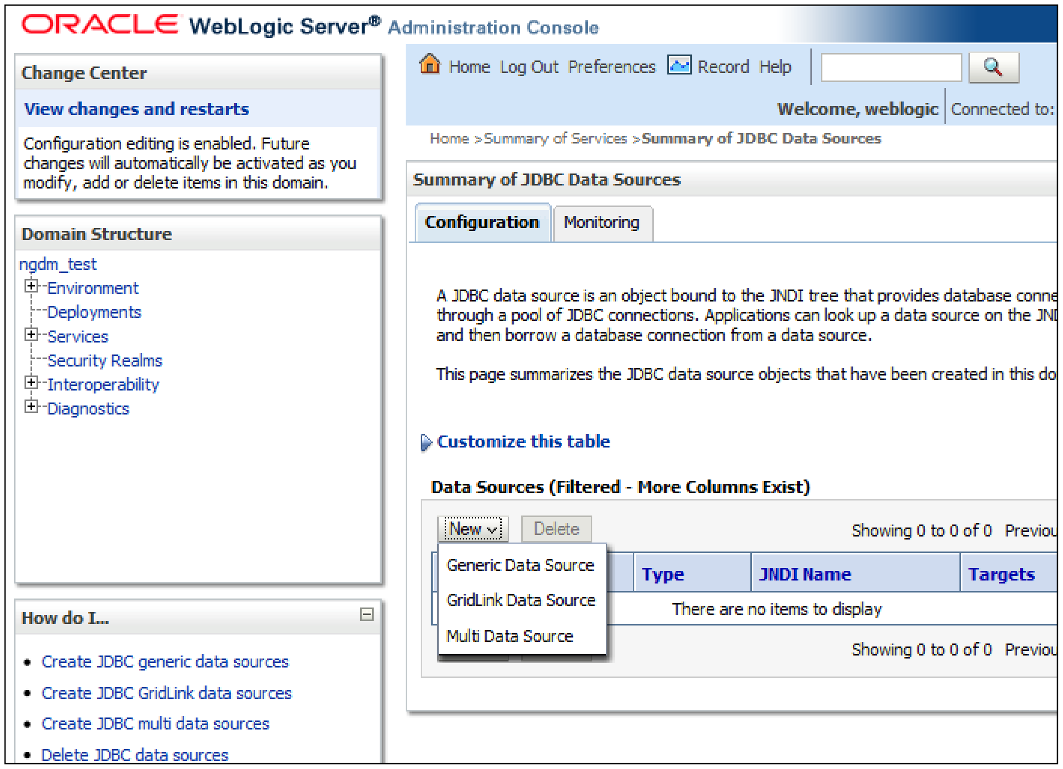
Environment
- Servers
- Clusters
- Virtual Hosts
- Migratable Targets

Services
- Messaging
  - JMS Servers
  - Store-and-Forward Agents
  - JMS Modules
  - Path Services

Interoperability
- WTC Servers
- Jolt Connection Poo

Diagnostics
- Log Files
- Diagnostic Modules
- Diagnostic Images

Click **Services** and then **Data Sources** to bring up the **Summary of JDBC Data Sources** page. Click **New | Generic Data Source** to define a new data source:


ORACLE WebLogic Server® Administration Console
Home Log Out Preferences Record Help
Welcome, weblogic Connected to:
Change Center
View changes and restarts
Domain Structure
ngdm_test
Environment
Deployments
Services
Security Realms
Interoperability
Diagnostics
How do I...
Create JDBC generic data sources
Create JDBC GridLink data sources
Create JDBC multi data sources
Delete JDBC data sources
Home >Summary of Services >Summary of JDBC Data Sources
Summary of JDBC Data Sources
Configuration
Monitoring
Customize this table
Data Sources (Filtered - More Columns Exist)
New
Delete
Generic Data Source
GridLink Data Source
Multi Data Source
Type
JNDI Name
Targets
Showing 0 to 0 of 0 Previou

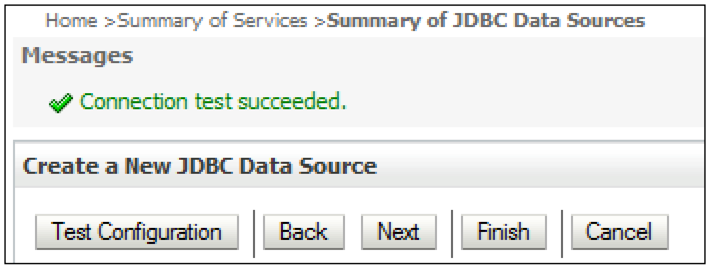
Configuration editing is enabled. Future changes will automatically be activated as you modify, add or delete items in this domain.

A JDBC data source is an object bound to the JNDI tree that provides database conne through a pool of JDBC connections. Applications can look up a data source on the JNI and then borrow a database connection from a data source.

This page summarizes the JDBC data source objects that have been created in this do

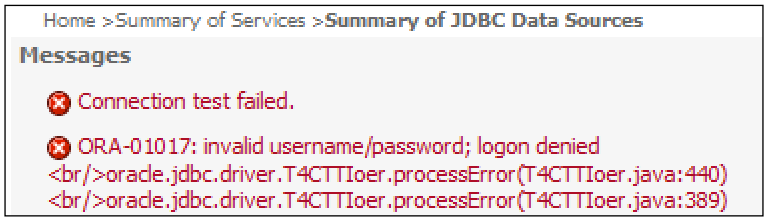
There are no items to display

Give your DataSource a name matching the one you used in your application module (for example, `XdmDS`) and a JNDI name also matching your application module (for example, `jdbc/XdmDS`). Leave database at **Oracle** and click **Next**. As your development database is unlikely to use **Real Application Clusters** (**RAC**), you can leave **Database Driver** at the default and click **Next**. Provide connection information (database, server, port, username, and password) and click **Next**. In the final step of the wizard, be sure to click the **Test Configuration** button. You should see a message that the connection test succeeded:


Home >Summary of Services >Summary of JDBC Data Sources
Messages
Create a New JDBC Data Source
Test Configuration
Back
Next
Finish
Cancel

Connection test succeeded.

If there is something wrong with your connection information, you get an error message instead:



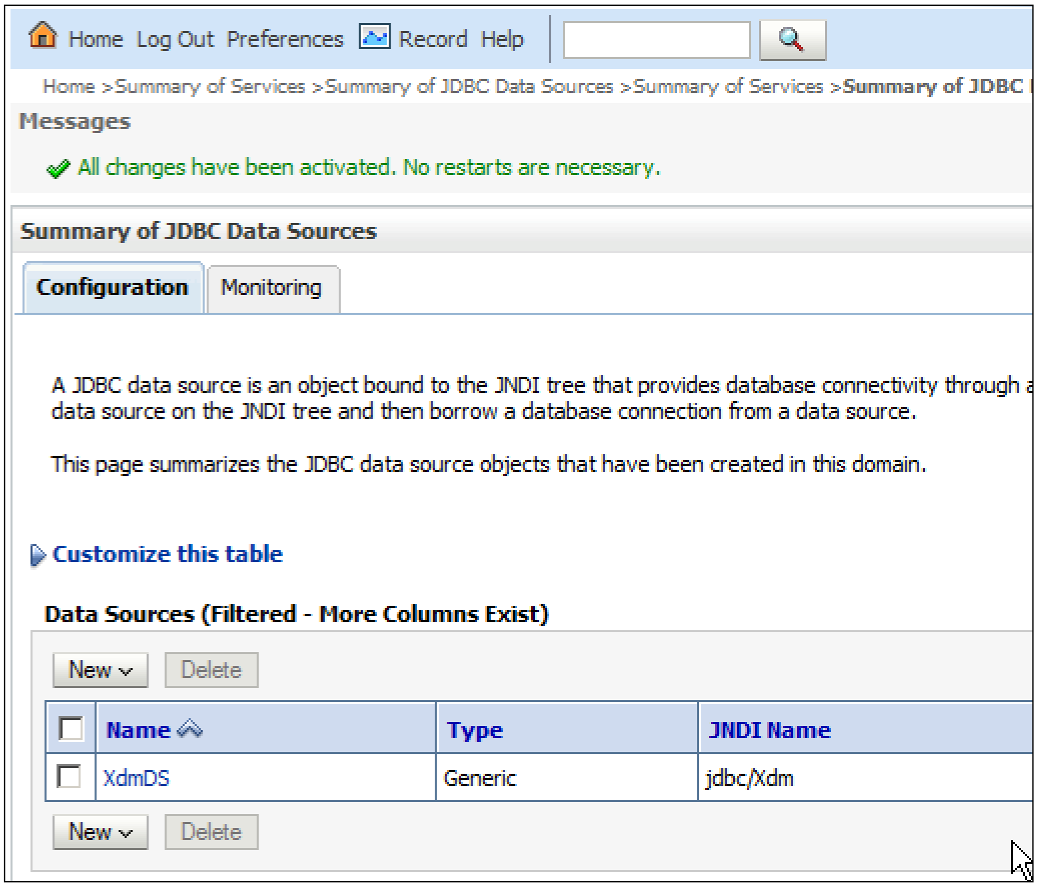
Home >Summary of Services >**Summary of JDBC Data Sources**

Messages

Connection test failed.

```
ORA-01017: invalid username/password; logon denied
<br/>oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:440)
<br/>oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:389)
```

Click **Finish** to close the wizard. You'll see your new data source on the **Summary** page:



Home >Summary of Services >Summary of JDBC Data Sources >Summary of Services >**Summary of JDBC**

Messages

All changes have been activated. No restarts are necessary.

Summary of JDBC Data Sources

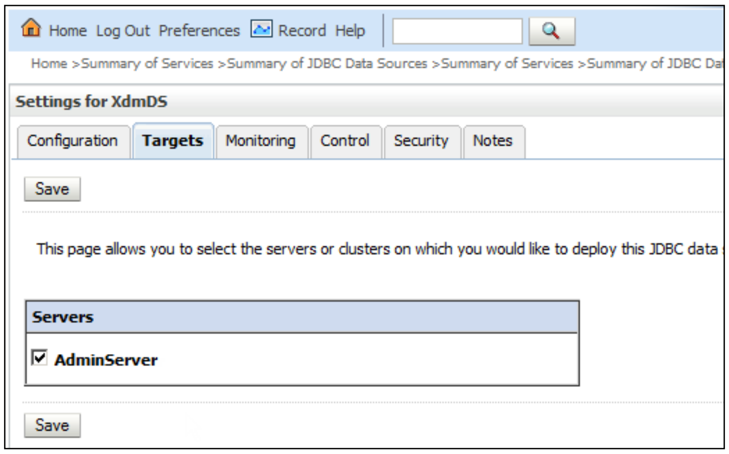
A JDBC data source is an object bound to the JNDI tree that provides database connectivity through a
data source on the JNDI tree and then borrow a database connection from a data source.

This page summarizes the JDBC data source objects that have been created in this domain.

Data Sources (Filtered - More Columns Exist)

☐	Name	Type	JNDI Name
☐	XdmDS	Generic	jdbc/Xdm

Note that your connection does not automatically acquire a target—the **Targets** column is blank. You need to click on your connection, choose the **Targets** tab and check the checkbox next to your server name:


Home Log Out Preferences Record Help
Home >Summary of Services >Summary of JDBC Data Sources >Summary of Services >Summary of JDBC Da
Configuration
Targets
Monitoring
Control
Security
Notes
Save

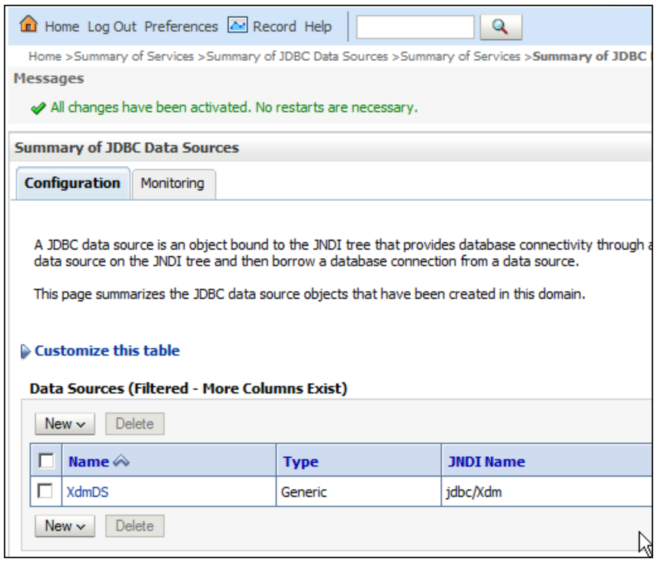
Settings for XdmDS

This page allows you to select the servers or clusters on which you would like to deploy this JDBC data

Servers
☑ AdminServer

On the simple development server you are managing in the development team, you only need the AdminServer. On the servers that your operations people manage, there is going to be one AdminServer, managing multiple **managed servers** inside the same WebLogic instance.

When you have saved your configuration, you should see your JDBC connection being associated with your server:


Home Log Out Preferences Record Help
Home >Summary of Services >Summary of JDBC Data Sources >Summary of Services >Summary of JDBC
Configuration
Monitoring
Customize this table
New
Delete

Messages

✓ All changes have been activated. No restarts are necessary.

Summary of JDBC Data Sources

A JDBC data source is an object bound to the JNDI tree that provides database connectivity through a data source on the JNDI tree and then borrow a database connection from a data source.

This page summarizes the JDBC data source objects that have been created in this domain.

Data Sources (Filtered - More Columns Exist)

	Name	Type	JNDI Name
☐	XdmDS	Generic	jdbc/Xdm

Deploying the application

With all of the development scaffolding removed from your application, and the application server prepared, it is time to deploy the application. Initially, you will do this directly from JDeveloper to the stand-alone development server, but the end goal is to deploy your application to an EAR file you can hand over to an application server administrator to install.

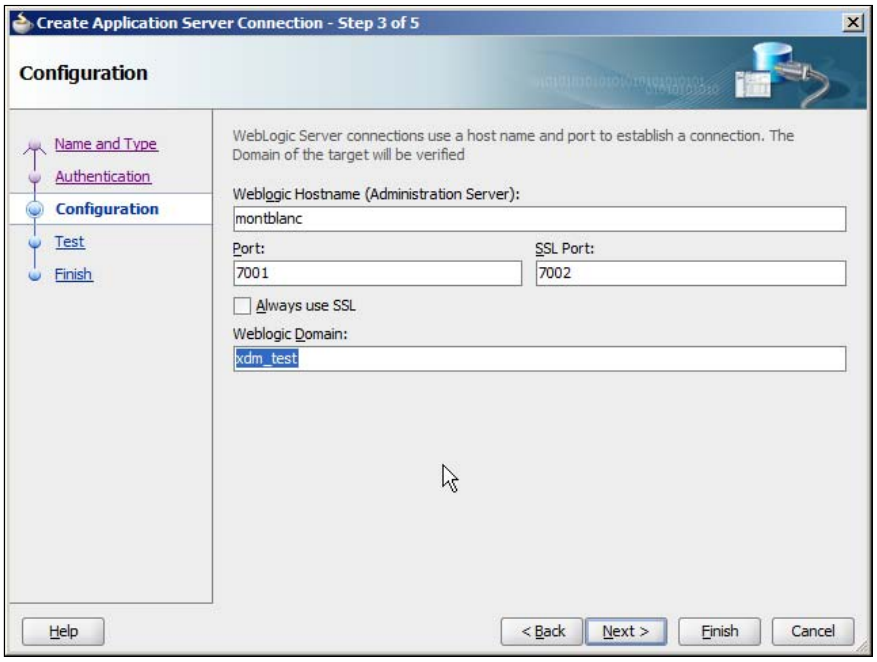
> If you are building a new database or have made any changes to an existing one, you also need to deliver matching database scripts.

Direct deployment

For the first test of your deployment procedure, you should deploy your application directly from JDeveloper to the application server. This approach gives you more feedback and makes it easier to fix any deployment errors. This is also the approach you normally use to deploy to the stand-alone development server that the development team *owns*.

Creating an application server connection

To deploy your application directly from JDeveloper, you first need to create an application server connection to your test/integration server. Choose **File** | **New** and then **General** | **Connections** | **Application Server Connection** to start the **Create Application Server Connection** wizard. In step 1, give your connection a name (my development server is called Montblanc, so I use `MontblancDev`), leave **Connection Type** at **WebLogic 10.3** and click **Next**. In step 2, provide a username and password for an administrator account on the server (the default username is **weblogic** and the default password is **weblogic1**). In step 3, fill in the host name of your server and the name of the WebLogic domain where you want to deploy your application. You created this domain when you installed the server:



Create Application Server Connection - Step 3 of 5

Configuration

WebLogic Server connections use a host name and port to establish a connection. The Domain of the target will be verified

Weblogic Hostname (Administration Server): montblanc

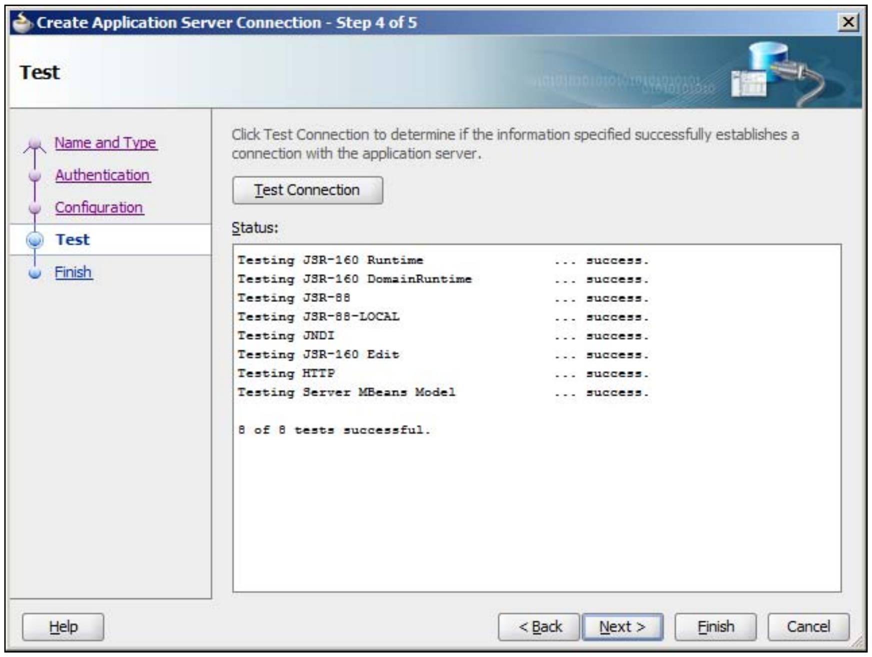
Port: 7001

SSL Port: 7002

Always use SSL

Weblogic Domain: xdm_test

In step 4, test that your connection works. All eight tests should pass:



Create Application Server Connection - Step 4 of 5

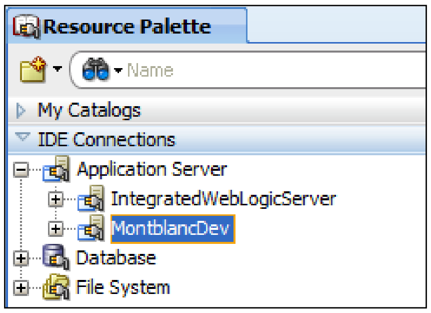
Test

Click Test Connection to determine if the information specified successfully establishes a connection with the application server.

Status:

```
Testing JSR-160 Runtime              ... success.
Testing JSR-160 DomainRuntime        ... success.
Testing JSR-88                       ... success.
Testing JSR-88-LOCAL                 ... success.
Testing JNDI                         ... success.
Testing JSR-160 Edit                 ... success.
Testing HTTP                         ... success.
Testing Server MBeans Model          ... success.

8 of 8 tests successful.
```

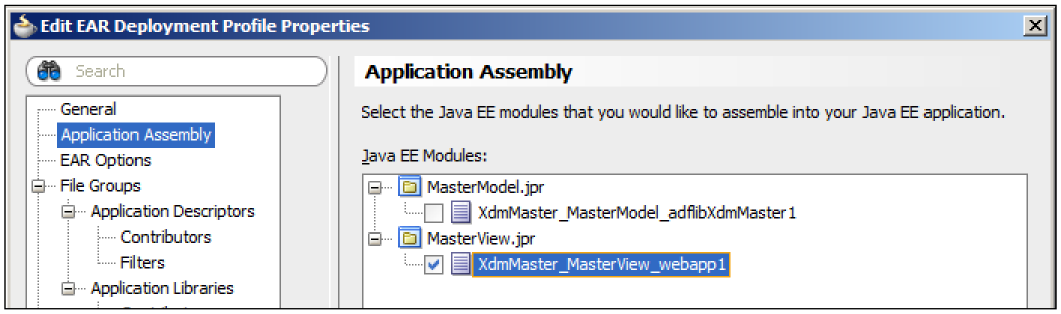
When you click **Finish**, your connection should show up under **IDE connections** in the **Resource Palette** in the upper-right part of the JDeveloper window (where the **Component Palette** also lives):



Deploying your application directly

You have already seen the deployment profile window earlier in this chapter when we deselected the checkbox to automatically create DataSources. This window is also where you create the deployment profile that controls how the application is deployed.

Choose **Application** | **Application Properties** and choose **Deployment** to bring up this dialog again. Delete any default application deployment profile you might find and click **New** to create a new deployment profile. Set **Archive Type** to **EAR File** and give your deployment profile a name (typically the same as your master workspace name—for the XDM application, this would be **XdmMaster**). In the **Edit EAR Deployment Profile Properties** dialog, choose the **Application Assembly** node and check the checkbox for your **MasterView** project:



Application Assembly

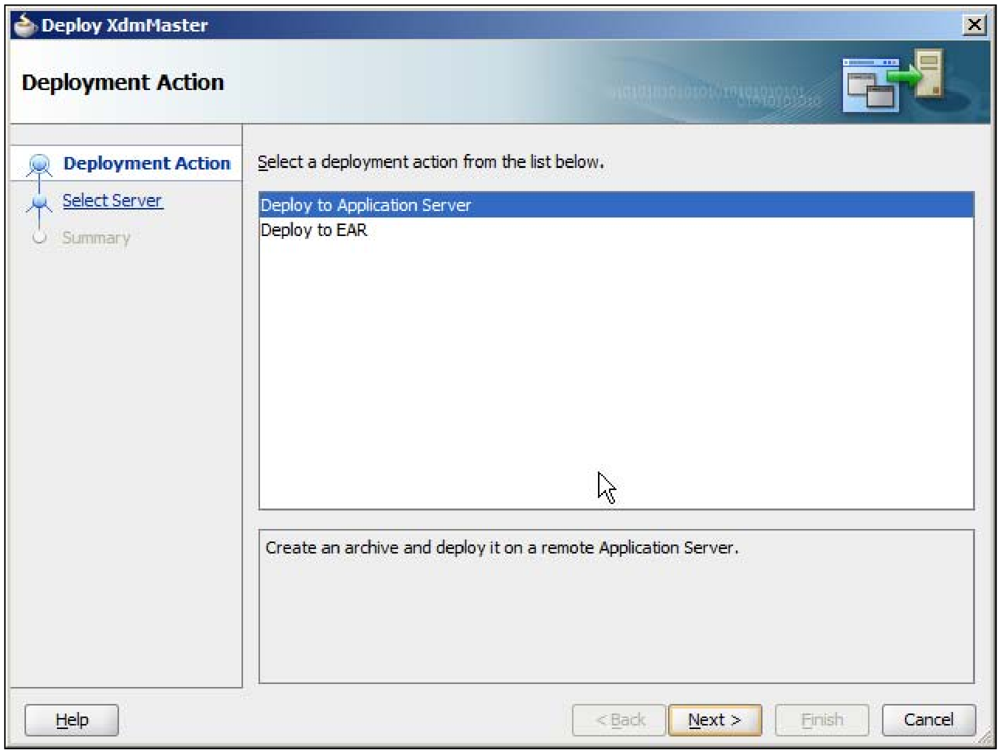
Select the Java EE modules that you would like to assemble into your Java EE application.

Java EE Modules:

- MasterModel.jpr
  - [ ] XdmMaster_MasterModel_adflibXdmMaster1
- MasterView.jpr
  - [✓] XdmMaster_MasterView_webapp1

You do not need to change any other settings in this dialog and can click **OK** to return to your application.

Now, you can choose **Application | Deploy** and choose the name of your application deployment profile (**XdmMaster**). You get a dialog box where you can choose a **Deployment Action**—choose **Deploy to Application Server** and click **Next**. In the next step, choose the application server connection you just created:



Select a deployment action from the list below.

Deploy to Application Server
Deploy to EAR

Create an archive and deploy it on a remote Application Server.

Click **Next** through the rest of the deployment wizard to start your deployment. This process first builds a Web Archive (WAR file) and then packages it into an Enterprise Archive (EAR file). It then calls the application server using the connection information you defined and installs your application on the application server.

You should see a new **Deployment** tab appear at the bottom of the screen, containing a series of deployment messages as follows:

```
[09:28:28 AM] ----  Deployment started.  ----
[09:28:28 AM] Target platform is  (Weblogic 10.3).
[09:28:28 AM] Retrieving existing application information
[09:28:28 AM] Running dependency analysis...
[09:28:28 AM] Building...
[09:28:29 AM] Deploying 2 profiles...
[09:28:29 AM] Wrote Web Application Module to C:\JDev …
```

```
[09:28:29 AM] Wrote Enterprise Application Module to …
[09:28:29 AM] Redeploying Application...
[09:28:39 AM] [Deployer:149194]Operation 'deploy' on …
[09:28:39 AM] Application Redeployed Successfully.
[09:28:39 AM] The following URL context root(s) were …
[09:28:39 AM] http://192.168.138.44:7001/NgdmMasterApp
[09:28:39 AM] Elapsed time for deployment:  10 seconds
[09:28:39 AM] ----  Deployment finished.  ----
```

If you do not get any error messages, you should now be able to start your application from your test server with a URL such as the one you see when running it locally. If you deploy the XDM application to a stand-alone development server called `montblanc` in the default managed server running on port `7001`, the URL to start the application would be `http://montblanc:7001/xdm/faces/Xdm.jspx`.

Remember that `xdm` is the **context root** of your application, set under **Project Properties | Java EE Application**. `Xdm.jspx` is the name of the jspx page you created in the master application as the starting point for your application.

File deployment through the console

Direct deployment is fine for testing your deployment procedure and deploying to the development server, but the package that you need to deliver to the operations people to install is an EAR file.

Creating the EAR file

To do this, again choose **Application | Deploy** and the name of your application deployment profile. In the deployment action dialog, choose **Deploy to EAR** and click **Next** and then **Finish**.

In the **Deployment** tab at the bottom of the screen, you will see a shorter series of deployment messages as follows:

```
[11:38:37 AM] ----  Deployment started.  ----
[11:38:37 AM] Target platform is  (Weblogic 10.3).
[11:38:37 AM] Running dependency analysis...
[11:38:37 AM] Building...
[11:38:40 AM] Deploying 2 profiles...
[11:38:41 AM] Wrote Web Application Module to C:\JDev …
[11:38:41 AM] Wrote Enterprise Application Module to …
[11:38:41 AM] Elapsed time for deployment:  4 seconds
[11:38:41 AM] ----  Deployment finished.  ----
```

In the line `Wrote Enterprise Application Module...`, you can see where JDeveloper wrote the `.ear` file. If your master application workspace is stored in `C:\JDeveloper\mywork\XdmMaster`, you can find the `XdmMaster.ear` file in `C:\JDeveloper\mywork\XdmMaster\deploy`.

> **What is in the file?**
> As mentioned earlier, an `.ear` file produced by JDeveloper is a normal compressed file that you can open with your decompression program of choice to peek into the file.

Deploying the EAR file

To deploy the EAR file you just created to your stand-alone development server, bring up the WebLogic console in your web browser again (the URL is your test/integration server name and port, followed by `/console`). The deployment procedure uploads your EAR file from your local file system to the server file system and then installs the application.

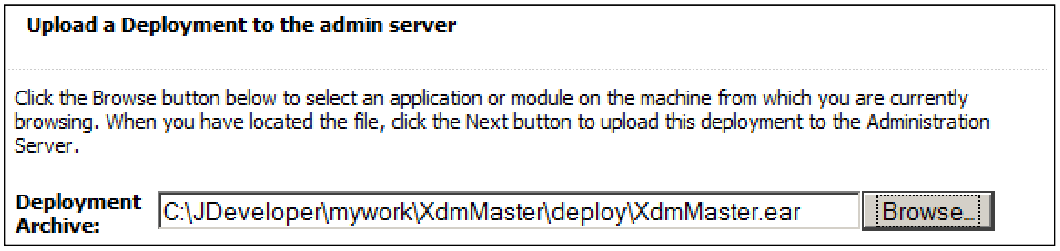
> **Production deployment**
> The procedure in this section can be used by your operations department to deploy the application to test, pre-production, and production servers if your organization is not using Oracle Enterprise Manager. If they are using Oracle Enterprise Manager, this software also has functionality to deploy and manage ADF applications.

Click **Deployments** in the **Domain Structure** box to the left. You will see quite a few deployments already; this is all the infrastructure that you do not need to worry about. You will also see the application you deployed directly from JDeveloper—in order to test deployment through the EAR file, you will need to select this and click **Delete**. Then click **Install** to start installing your application from the EAR file.

In the first step of the **Install Application Assistant**, you need to point to your EAR file. Remember that you just built your EAR file on your local machine, and you are now running the console application on your test/integration server. This means that you have to click the **upload your file(s)** link to copy your EAR file from your development machine to the server.

> **Separate file systems**
> Your EAR file is generated on your local machine, but the content of the **Path** field in the WebLogic console refers to file locations on the test/integration server. Unless you generated your EAR file to a network drive accessible to the test/integration server, you have to upload your EAR file.

In the next step of the wizard, click the **Browse** button to select your local EAR file:

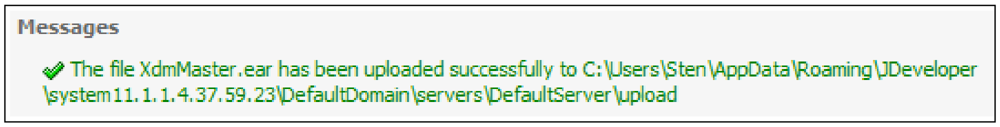


Upload a Deployment to the admin server

Click the Browse button below to select an application or module on the machine from which you are currently browsing. When you have located the file, click the Next button to upload this deployment to the Administration Server.

Deployment Archive: C:\JDeveloper\mywork\XdmMaster\deploy\XdmMaster.ear

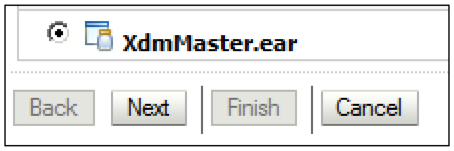
You should get a message that your EAR file was successfully uploaded. This means that the file is now present in the file system on the application server:



Messages

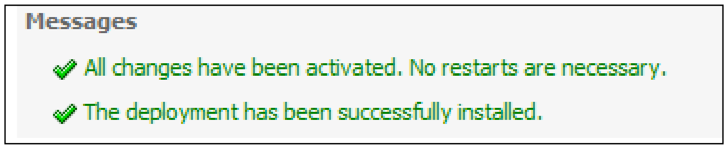
✓ The file XdmMaster.ear has been uploaded successfully to C:\Users\Sten\AppData\Roaming\JDeveloper\system11.1.1.4.37.59.23\DefaultDomain\servers\DefaultServer\upload

It should also be listed at the bottom of the screen for you to select:



XdmMaster.ear

If you have uploaded several files for different applications, the console web page might list multiple files. Select the one you just uploaded and click **Next** to start the actual deployment. Choose to install this deployment as an application and click **Next**. Then give your application a name and click **Next**. In the final step of the installation assistant, you are presented with a number of optional settings—you can leave these at the default settings for a simple deployment to your test/integration server. Click **Finish** to start the deployment. After a little while, you should receive a message saying your deployment has been successfully installed:



Messages

✓ All changes have been activated. No restarts are necessary.

✓ The deployment has been successfully installed.

As when deploying directly, your application should now be available on the test/integration server with the same URL.

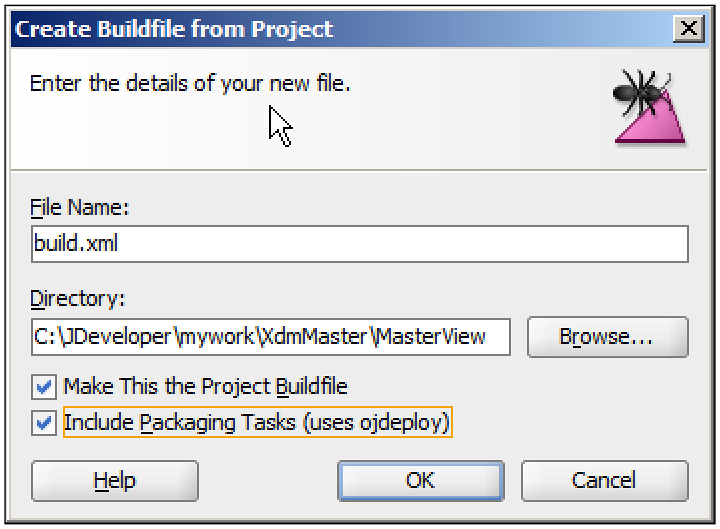
Scripting the build process

During the project, you will be building and deploying many times, so makes good sense to create scripts that handle this whole process. A good tool for handling this scripting is Apache Ant (`http://ant.apache.org`).

Creating a build task

When working with Ant, you create a **buildfile** (traditionally called `build.xml`) to specify how to build a project. This XML file consists of a number of **targets** that define the different goals you might want your build process to achieve, for example `clean`, `init`, `compile`, `test`, or `deploy`. Within each target are a number of steps called **tasks**. Ant comes with a large number of pre-built tasks, and many tools that integrate with Ant supply their own tasks. If you are not already familiar with Ant, there are several books and many online resources available, for example, the online manual at `http://ant.apache.org/manual/index.html`.

JDeveloper offers to help you create the Ant build files for your project. To do this, you first need to add Ant to the technologies in the project. Right-click your master view project (for example, **MasterView**) and **Project Properties**. Under **Technology Scope**, select **Ant** and move it to the right-hand **Selected** box. Then click **OK** to return to your project. Now choose **File** | **New** and then **Ant** (under **General**). Choose **Buildfile from Project** and click **OK**. The **Create Buildfile from Project** dialog is shown:


Create Buildfile from Project
Enter the details of your new file.
Browse...
Help
OK
Cancel

File Name: build.xml

Directory: C:\JDeveloper\mywork\XdmMaster\MasterView

[✓] Make This the Project Buildfile

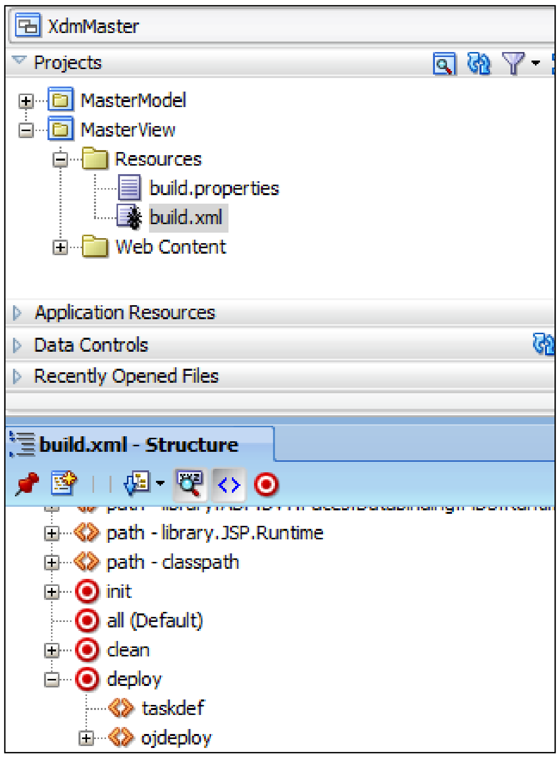
[✓] Include Packaging Tasks (uses ojdeploy)

It is important that you check the checkbox **Include Packaging Tasks (uses ojdeploy)**—this tells JDeveloper to also generate targets and tasks for packaging and deploying your application.

JDeveloper deployment without the User Interface

The *ojdeploy* referred to by the dialog box is a command-line java program that can do anything JDeveloper can do with regards to deployment. This program is included with JDeveloper so that you can automate your build process as described in this section. It uses some JDeveloper libraries, so you need JDeveloper or a special JDeveloper library installed on the machine where you run it. Refer to the documentation for a thorough explanation of the many options with `ojdeploy`.

When you click **OK**, you will see in the **Application Navigator** that two new files were created for you: the **build.xml** file defining how to build the project, and the **build.properties** file containing some constants such as, directory names, and so on. You will need to modify the properties file to match your local environment. The little insect icon on the **build.xml** file shows that JDeveloper has recognized the file as an Ant build file:



The **Structure** panel at the bottom-left of the JDeveloper window will also show targets and tasks in your build file.

You can right-click on an Ant build file and run individual targets from within the JDeveloper.

Moving your task to the test/integration server

When you have tested your Ant script, you need to move it to your own stand-alone development server in order to enable the automatic build on this server. You will also have to install the Ant software on the server. Additionally, you will need the `ojdeploy` runtime code.

JDeveloper provides the necessary libraries to run `ojdeploy`, so that is the easiest option. However, it is also possible to install the libraries without placing JDeveloper itself on your server. At the time of writing, an explanation of how to achieve this was documented in the *Oracle ADF Essentials* series of articles by John Stegeman. You can find these on the Oracle Technology Network (`http://otn.oracle.com`, search for *adf essentials*).

Once you have Ant and the `ojdeploy` libraries on your stand-alone development server, copy the `build.xml` and `build.properties` files to the server. After copying, you need to open the `build.properties` file and change it to match your server environment. This file is a simple collection of keys and values, as follows:

```
oracle.jdeveloper.ant.library=
    C\:\\oracle\\Middleware11.1.1.4\\jdeveloper\\jdev\\
    /lib/ant-jdeveloper.jar
output.dir=classes
oracle.home=../../../../oracle/Middleware11.1.1.4/jdeveloper/
javac.deprecation=off
oracle.jdeveloper.workspace.path=
    C\:\\JDeveloper\\mywork\\NgdmMasterApp\\NgdmMasterApp.jws
```

All of the path values need to be changed to match your test/integration server environment.

Adding a Checkout

Now we are able to run the build process on the test/integration server, but we do not have any code to build on the server yet. That is where the Ant <svn> task comes in.

We do not want our build process to be based on what some developer happens to have on his development workstation—we want to build from the checked-in code in the Subversion repository.

Ant does not come with Subversion integration built in, but as mentioned previously, it is easy for other projects to program and deliver Ant tasks. For integrating Subversion with Ant, you can use the SvnAnt task available from `http://subclipse.tigris.org/svnant.html`. This task depends on having access to a Subversion command-line client; if your Subversion installation did not include this, there are several options available (for example, the CollabNet client from `http://www.collab.net` or the Slik SVN client from `http://www.sliksvn.com`).

With this software installed on your test/integration server, you can add an additional target to your `build.xml` file for checking out the source code, using a `<svn>` task. Your code might look something like the following:

```
<svn username="${username}" password="${password}">
  <checkout url="http://montblanc:8088/svn/repos/xdm/trunk"
    revision="HEAD" destPath="build" />
</svn>
```

If you add your `checkout` task to the `depends=` property of your deploy target, Ant will automatically check out the latest code before running `ojdeploy` to package and deploy your application to the WebLogic server.

Adding the database

If your enterprise ADF application includes any changes to the database (new or modified objects), you can use Ant `<sql>` tasks to run these scripts as part of your build process.

More scripting

Once you have your basic build script, you can add many other things:

- You can tag each nightly build in Subversion by using the `<copy>` operation to copy the revision you check out to your subversion `/tags` directory
- You can run your JUnit unit tests as part of the build with `<antunit>`
- You can run your Selenium user interface tests as part of the build with `<selenese>`

Automation

Your build script ensures that a developer only has to issue one command to build the whole application, and that nothing is left out or forgotten.

However, scripting is only the first step towards true automation. Because all of your tasks can now be started by another program, you can begin using continuous integration tools such as Hudson (`http://hudson-ci.org`) to automatically run nightly builds, or even start a new build every time a developer checks code in.

Summary

The XDM application has been cleaned up and the parameters checked for deployment. You have set up your stand-alone development server and checked that you can package, deploy and run the XDM application. Together with database scripts and installation instructions, you are now ready to deliver a deployment package to the operations team to install on the test environment. You have also seen how you can use Apache Ant to script this process.

When your test team has completed testing, and you have corrected any issues the test has uncovered, you use the same procedure to create a new final deployment package that your operations team will install on the production server.

Your ADF enterprise application is ready for business.

Internationalization

In his book "*The Hitchhiker's Guide to the Galaxy*", Douglas Adams describes a world where people can place a "Babel fish" in their ears to instantly understand any language. And while the real-life "Babel fish" at `http://babelfish.yahoo.com` or `http://translate.google.com` does a fair job of automatic translations, an enterprise application intended for people speaking different languages still has to be translated, at least, in part by humans.

Even if you are only planning to run your application in one language, please read this appendix to see how easy it is to internationalize your application. If you do the internationalization while you develop the application, it is easy; if you have to do it later, it is hard.

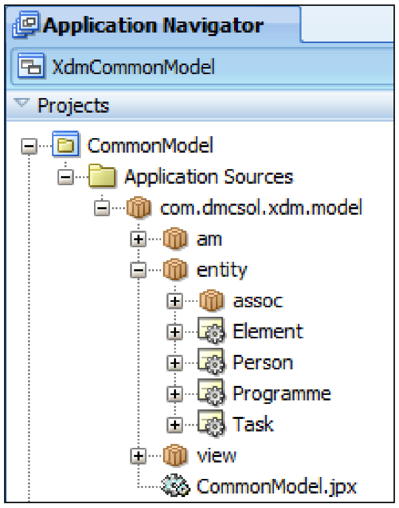
Localization lingo

Internationalization means building your application so it can be adapted to different languages and countries. People who work in this area often shorten this very long word to just **i18n** (eighteen characters in the middle of the word removed).

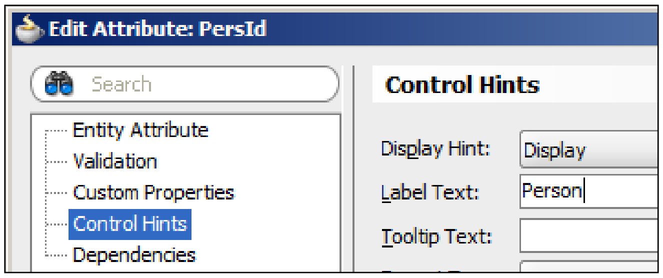
Localization means actually preparing your application for a specific language and country. This will involve translating the user interface text, but also changing date formats and making other changes for the specific country. This is sometimes abbreviated **L10n**.

Automatic internationalization

JDeveloper is built for enterprise applications, so it automatically prepares your application for localization. Let's take the XDM Common Model as an example:



Now watch what happens when we go into the **Task** entity object and define a *Control Hints* for an attribute:

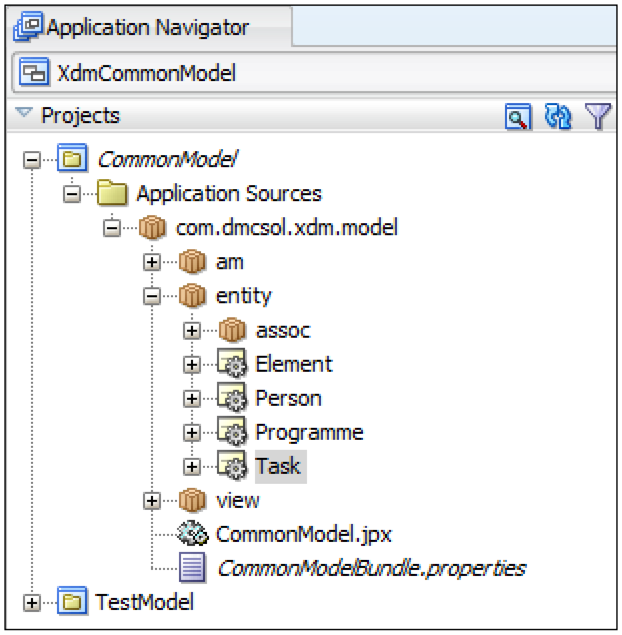


Display Hint: Display

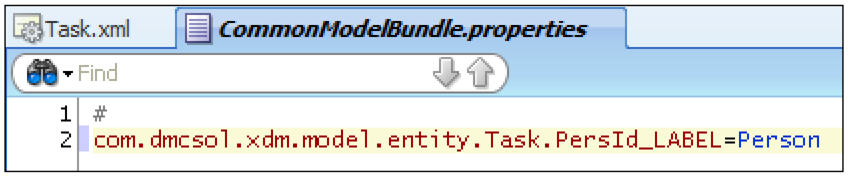
Label Text: Person

Tooltip Text:

Because the `PersId` is something that might be shown to the application end user—for example as a prompt for a drop-down list—JDeveloper does not just hardwire the literal string into the application. Instead, JDeveloper automatically creates a **Resource Bundle** for you. You can see this new file in the **Application Navigator**:



Notice the new file **CommonModelBundle.properties**. If you open this file, you'll see something like this:

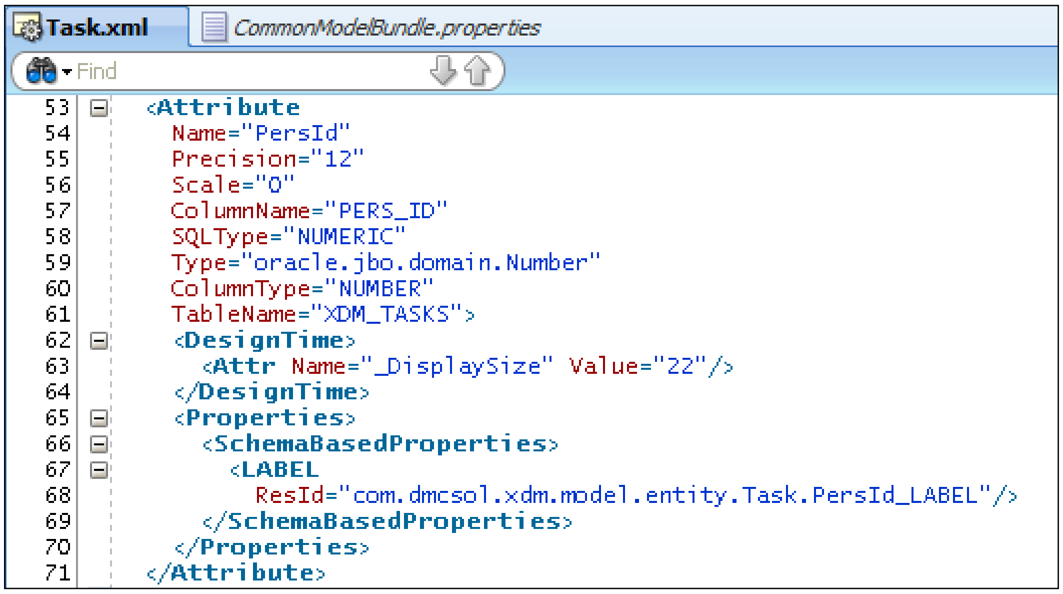


```
#
com.dmcsol.xdm.model.entity.Task.PersId_LABEL=Person
```

JDeveloper has automatically:

- Extracted the value you defined as the label control hint for the attribute
- Created a Resource Bundle file
- Placed the text you entered into the Resource Bundle and assigned a key to it
- Inserted a reference to the resource key into the entity object

If you click the **Source** tab for the entity object, you can see that the **ResId** attribute for the label points to **com.dmcsol.xdm.model.entity.Task.PersId_LABEL**—the key that JDeveloper automatically created in the resource bundle:



```
<Attribute
  Name="PersId"
  Precision="12"
  Scale="0"
  ColumnName="PERS_ID"
  SQLType="NUMERIC"
  Type="oracle.jbo.domain.Number"
  ColumnType="NUMBER"
  TableName="XDM_TASKS">
  <DesignTime>
    <Attr Name="_DisplaySize" Value="22"/>
  </DesignTime>
  <Properties>
    <SchemaBasedProperties>
      <LABEL
        ResId="com.dmcsol.xdm.model.entity.Task.PersId_LABEL"/>
    </SchemaBasedProperties>
  </Properties>
</Attribute>
```

How localizable strings are stored

There are three ways to store localizable strings in an ADF application:

- In a simple `.properties` file
- In an XLIFF file (an XML file format)
- In a Java `ListResourceBundle` class

The example you just saw uses a simple `.properties` file, which is easiest to work with.

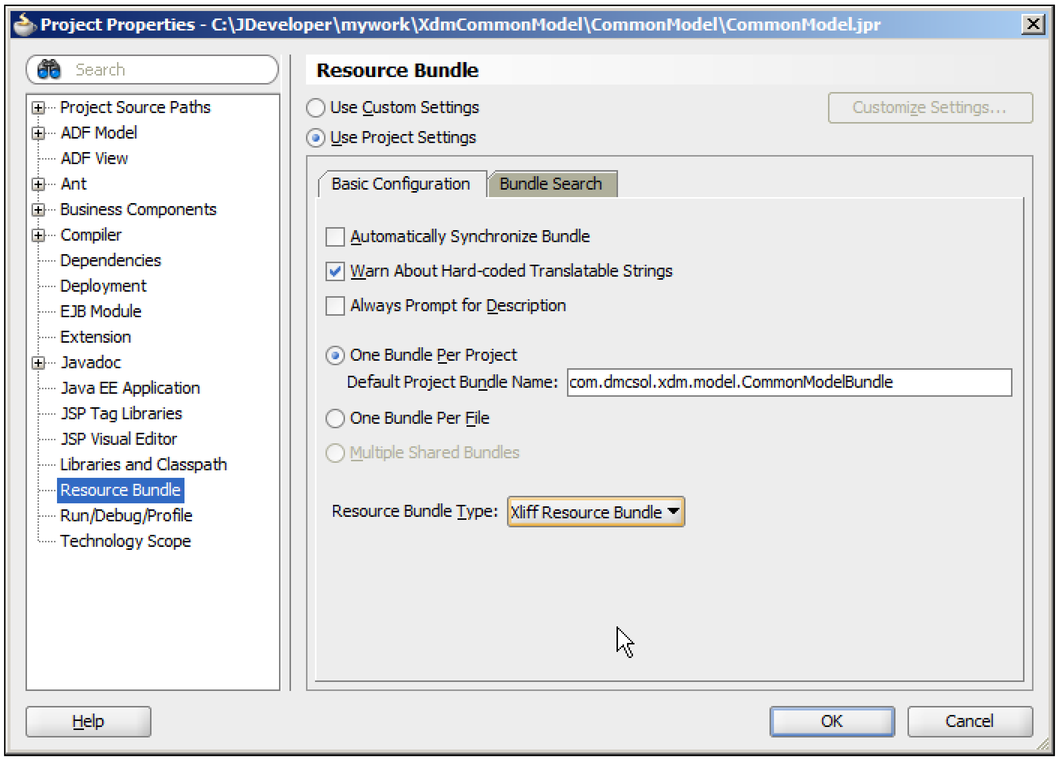
If you will be using a professional translation service to translate the user-visible text strings, they are likely to ask you for an XLIFF file. **XLIFF** stands for **XML Localization Interface File Format**, and professional translation software will be able to read and write XLIFF files. Oracle MetaData Services also uses XLIFF files to store customized strings as we saw in *Chapter 9, Customizing the Functionality*.

The last option is to define all your strings in a Java class that extends `java.util.ListResourceBundle`. This class must implement a method to return all the localizable strings in an `Object[][]` as follows:

```
package com.dmcsol.xdm.model;
import java.util.ListResourceBundle;
public class ModelBundle extends ListResourceBundle {
  private static final Object[][]contents =
  {
    { "com.dmcsol.xdm.model.entity.Task.PersId_LABEL",
      "Person" },
    { "com.dmcsol.xdm.model.entity.Task.StartDate_LABEL",
      "Start time" }
    { "com.dmcsol.xdm.model.entity.Task.StartWhere_LABEL",
      "Start location" }
  }

  public Object[][]getContents() {
    return contents;
  }
```

The class has to define an array with an element for each of your localizable strings. This element is again an array containing exactly two values: the key and the localized value. Because the `ListResourceBundle` is much more difficult to read and write, you typically do not use this if you are only using static strings. If you try to deliver a file like this to your localization team, you can be sure that the commas and curly brackets will not all be correct in the file you get back. However, it does make sense to use a `ListResourceBundle` if you plan to keep all your localizable strings in a database. In this case, your resource bundle class can access the database to retrieve the values.

You choose which way you want to store your localizable strings on a project-by-project basis under **Project Properties**. In this dialog, choose the **Resource Bundle** node on the left, and choose the desired **Resource Bundle Type** on the right. For the XDM application, we choose the XLIFF format because we expect the application to become an international success - and we expect to send out the application UI strings to a translation agency when that happens.


Project Properties - C:\JDeveloper\mywork\XdmCommonModel\CommonModel\CommonModel.jpr

Resource Bundle

Use Project Settings selected.

Basic Configuration tab:

- [ ] Automatically Synchronize Bundle
- [✓] Warn About Hard-coded Translatable Strings
- [ ] Always Prompt for Description

One Bundle Per Project selected.

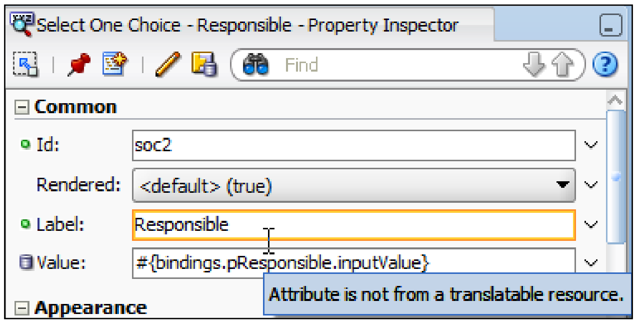
Default Project Bundle Name: com.dmcsol.xdm.model.CommonModelBundle

Resource Bundle Type: Xliff Resource Bundle

A binding choice

Once you have selected the resource bundle type, your project will use that type onward. If you go back into the properties dialog and change the resource bundle type after you have started using another type, you will need to delete the re-source bundle JDeveloper has started for you in order to start over with the new type.

It is a good idea to check the checkbox **Warn About Hard-coded Translatable Strings**—this tells JDeveloper to present you with a warning if you hard-code a string into a translatable field such as the **Label**. This will be shown with an orange border around the properties that should come from a resource bundle:



Id: soc2
Rendered: <default> (true)
Label: Responsible
Value: #{bindings.pResponsible.inputValue}
Attribute is not from a translatable resource.

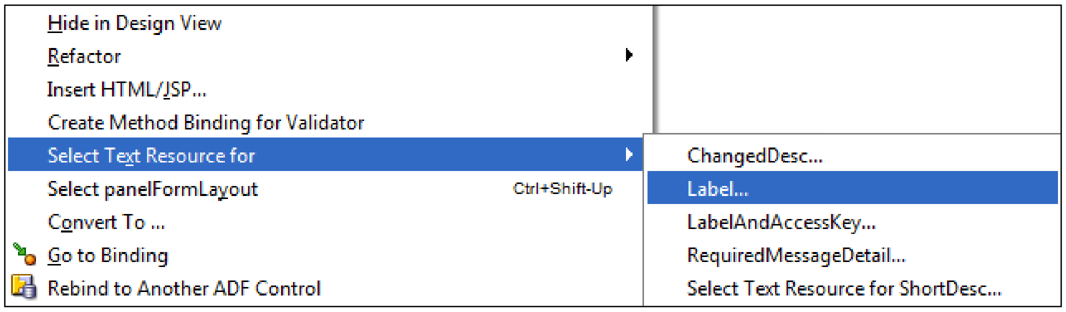
The warning will also be shown in the source view for a page or a business component.

Defining localizable strings

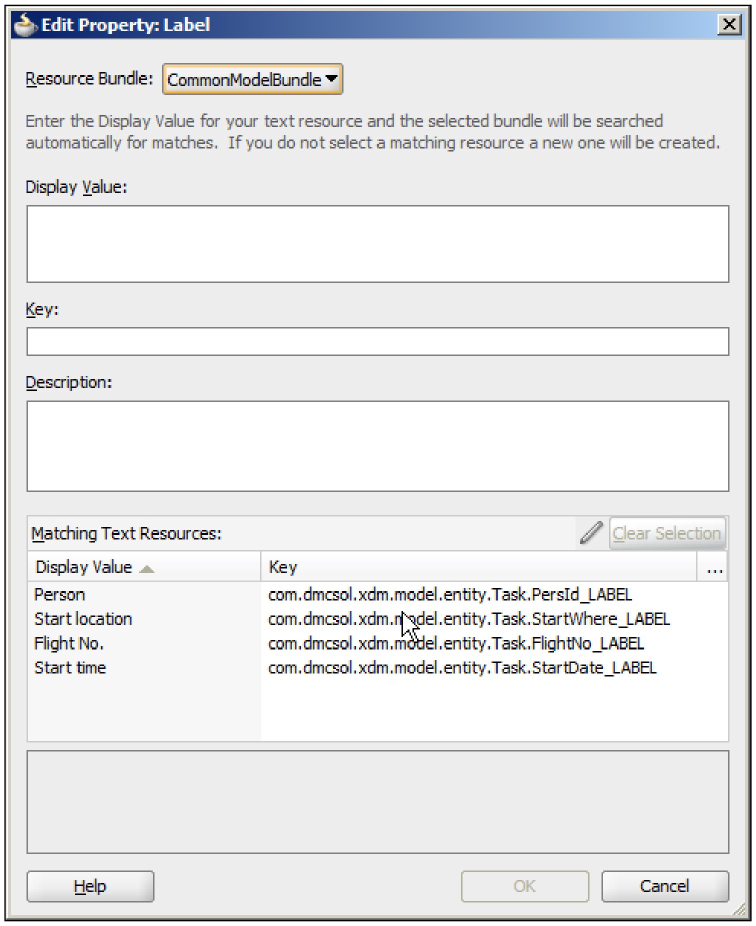
Every string the user sees can be localized - labels for fields, mouseover texts, validation messages, and so on.

In some cases (such as the entity attribute above), JDeveloper can automatically register a new string and create an associated key in a resource bundle. However, you will normally work in the Select Text Resource dialog to define your strings, because this dialog provides the option to select an already defined string for a new purpose. You can invoke this dialog from many places:

- To set a business component attribute (for example, **Label**), you click on the looking glass icon to the right of the field in the **Control Hints** dialog
- To set a text for a user interface element from the **Property Palette**, you can click on the little triangle to the right of the field and choose **Select Text Resource** from the pop-up menu
- To set a text for a user interface element from the visual page designer, right-click on the element and choose **Select Text Resource for** and then the element you want to define (for example, **Label**):



All of these bring up the **Select Text Resource** dialog, where you can define new strings or choose among the existing ones:


Edit Property: Label
Resource Bundle: CommonModelBundle
Enter the Display Value for your text resource and the selected bundle will be searched automatically for matches. If you do not select a matching resource a new one will be created.
Display Value:
Key:
Description:
Matching Text Resources:
Clear Selection
Help
OK
Cancel

Display Value	Key
Person	com.dmcsol.xdm.model.entity.Task.PersId_LABEL
Start location	com.dmcsol.xdm.model.entity.Task.StartWhere_LABEL
Flight No.	com.dmcsol.xdm.model.entity.Task.FlightNo_LABEL
Start time	com.dmcsol.xdm.model.entity.Task.StartDate_LABEL

As you start typing in the **Display Value** field, JDeveloper automatically fills in the **Key** field with a suggested key. At the same time, the **Matching Text Resources** box is automatically reduced to the elements matching the display text you are entering. If an already defined text resource contains the text you want, you can select it in this box and click **Select**. Otherwise, type the display value, accept or change the suggested key, and click **Save and Select** to store your new key/value pair in the resource bundle.

The **Description** field is optional unless you checked the checkbox **Always Prompt for Description** under the **Resource Bundle** in the **Project Properties** dialog.

Give us a clue

It is very hard for a translator to translate an individual word without any indication of the context where it is used. If you do not provide good descriptions, you will either be spending a lot of time answering questions from your translator, or a lot of time correcting language errors once you show your enterprise application to native speakers of your target language.

Performing the translation

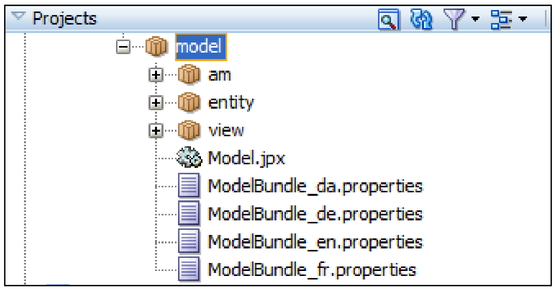
Now that you have your strings nicely separated out from business components and user interface, it is time to translate them. You have a resource bundle (`.properties` file, XLIFF file, or Java class) with your default text—but you have not really told the ADF framework what that language is. To add this information, you add a suffix to the file, using a two-character ISO 639 language code. If, for example, your default language is *English*, you add `_en` to the file name, making a file such as `ModelBundle_en.properties`.

If you wish to specify a specific country version of the language, you can add an additional suffix using a two-character ISO 3166 country code. For example, French as spoken in France would be `ModelBundle_fr_FR` while French as spoken in Canada would be `ModelBundle_fr_CA`.

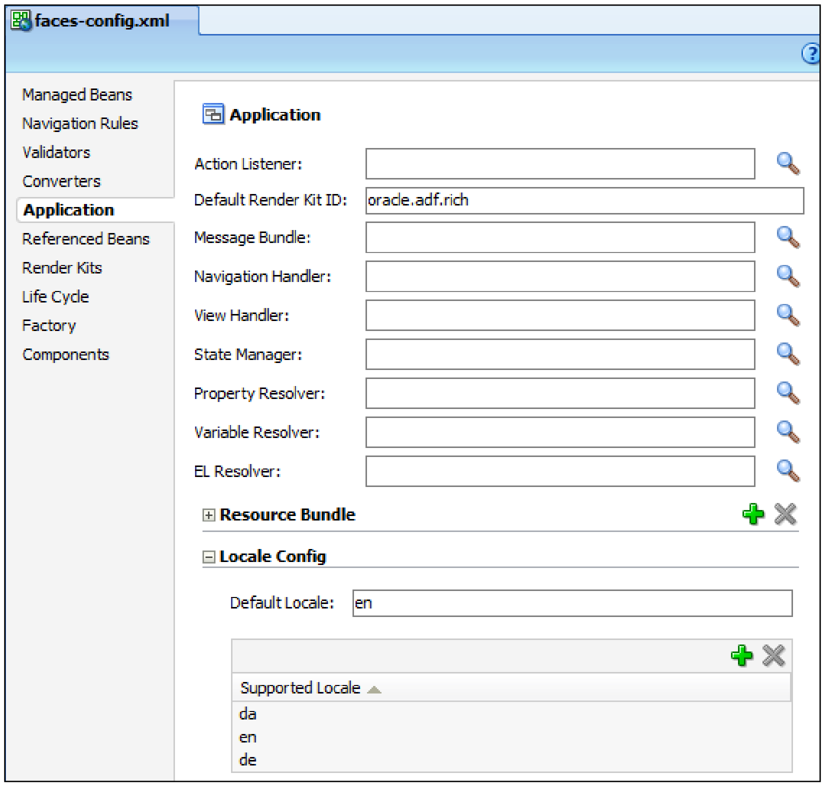
To start your translation process, you create copies of your default file or class with different suffixes for all your target languages. You can then send out your property or XLIFF file to be translated.

Do not send a Java `ListResourceBundle` class to be translated unless your translator happens to be a Java programmer in his spare time.

When you get your translation back, you need to place all of your translated resource files in the file system next to the original default resource files in your project:


Projects
model
am
entity
view
Model.jpx
ModelBundle_da.properties
ModelBundle_de.properties
ModelBundle_en.properties
ModelBundle_fr.properties

Additionally, you will have to define the languages your application will offer in the **faces-config.xml** file. You can find this file in your **View** project under **Web Content / WEB-INF**. On the **Application** sub-tab, scroll down to the **Locale Config** section, set **Default Locale** and add all the languages your application supports:



Default Render Kit ID: oracle.adf.rich

Default Locale: en

Supported Locale
da
en
de

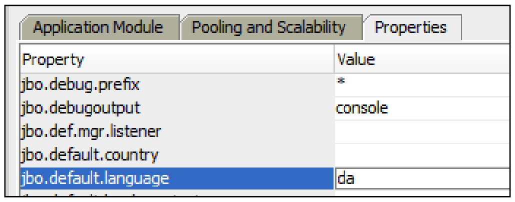
Running your localized application

With the translated files back and integrated into the application, it is time to test the localized versions.

Testing the localized business components

When you right-click on an application module to run it through the Business Component Browser, it will by default run in the language selected in your operating system. If you are running Microsoft Windows, the Business Component Browser will show the label control hint in the language you have selected in the Windows Control Panel (under **Region and Language**).

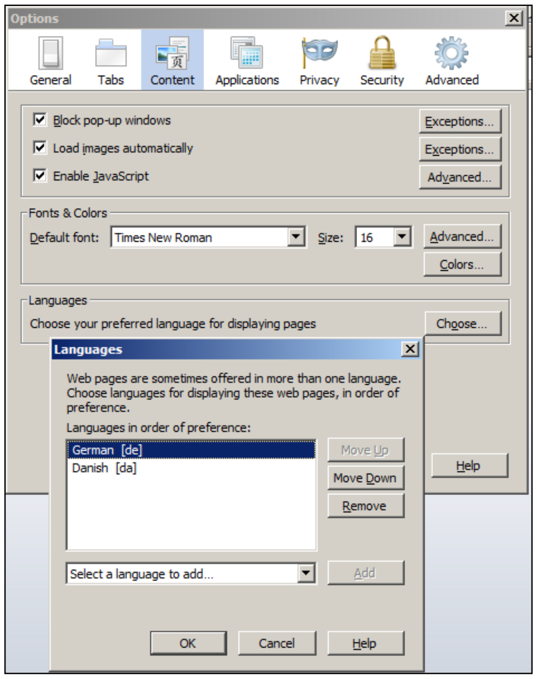
You can override this setting by setting the default language of your application module. This is done by right-clicking the application module, choosing **Configurations** and then **Edit**. On the **Properties** tab, scroll down to the **jbo.default.language** property and set a value to force the business component tester to show your application module in that language:



Property	Value
jbo.debug.prefix	*
jbo.debugoutput	console
jbo.def.mgr.listener	
jbo.default.country	
jbo.default.language	da

Testing the localized user interface

When you are running the application as the user will see it, you will be running web pages in a browser. It is part of the HTTP protocol that the browser will send an ordered list of the languages it would like to see content in, so the web server can serve up this content if available. You can set the language preferences in the browser—in Firefox 3.6, this setting is found under **Tools**, **Options**, **Content**. On this tab, you find a **Languages** section with a **Choose** button that opens the dialog to set language preferences:



☑ Block pop-up windows
☑ Load images automatically
☑ Enable JavaScript

Default font: Times New Roman Size: 16

Choose your preferred language for displaying pages

Languages

Web pages are sometimes offered in more than one language. Choose languages for displaying these web pages, in order of preference.

Languages in order of preference:

German [de]
Danish [da]

Select a language to add...

In Internet Explorer, this setting is found under **Tools | Internet Options**. On the **General** tab, click the **Languages** button to change your language preferences.

Note that the browser language settings determine a priority—so the previous setting means: "First ask for a German version of the page, then a Danish one." If your ADF application does not support any of the requested languages, it will revert to the default language defined in the `faces-config.xml` file.

To test your application in different languages, it is enough to change the order of the languages in your web browser and refresh the page—the page will re-draw in the first language your application supports.

Localizing formats

When you change language in your browser, you will see date and number formats change automatically as well. When for example running the application in Danish, Christmas Eve is displayed as **24.12.2011**, while when running it in English, the same date is shown as **12/24/2011**. Note that both the day/month order and the separator character have changed. If you have chosen a date format that includes month names, these will be localized as well.

ADF will also automatically change the decimal character between a period and a comma, matching the locale your browser is set to.

Do not use the currency format

You have the option to select **Currency** as **Format Type**. Don't use this format (or use it with extreme care)—ADF does not know exchange rates. This means that if you format something with a currency symbol and change your browser setting, the same amount might suddenly be displayed as Euros instead of U.S. Dollars. Use a normal number field and place the currency symbol in a label or prompt next to it.

More internationalization

The error messages you define for the business rules in your entity objects should also point to a string in a resource bundle instead of containing hard-wired error or warning messages.

If you have data in several languages in your tables and want to present it, for example, a list of values in a user language, you need a managed bean and a bind variable in your view object. The managed bean has access to the browser session variables, can retrieve the UI language, and store it in a variable. The view object can then use a bind variable assigned a value using the built-in `adf.context` object to refer to the bean value. Therefore, if you have a `LocaleHelperBean` with Session scope and a `userLanguage` parameter, you would assign your bind variable to the expression `adf.context.sessionScope.LocaleHelperBean.userLanguage`.

You will notice that the standard texts that ADF supplies (for example, "Sort Ascending" when pointing to table column header) will also be localized. ADF comes in all the languages Oracle normally localize their end user software to—several dozen at the last count. As we discussed in *Chapter 9*, *Customizing the Functionality*, these strings are part of the skin; if you are not happy with Oracle's standard texts in a language, you can create a skin that overrides them with your own.

If you need to change the language programmatically, this is possible as well. It is a bit of an advanced topic and falls outside the scope of this book, as it involves creating and registering a phase listener to ensure that the chosen language is always set before the page is rendered.

Summary

You have seen how to use resource bundles to ensure that your application can easily be localized into different languages to match the user's environment. You are ready to build ADF enterprise applications for the world!

Index

PACKT
PUBLISHING