Oracle9*i* Application Server

Application Developer's Guide

Release 2 (9.0.2)

January 2002

Part No. A95101-01

Oracle9*i* Application Server Application Developer’s Guide, Release 2 (9.0.2)

Part No. A95101-01

Contents

1 Creating Applications: Overview

1.1 Overview of Oracle9*i*AS 1-2
1.1.1 J2EE 1-2
1.1.2 Enterprise Portals 1-2
1.1.3 Wireless Support 1-3
1.2 Development Steps 1-5
1.3 Development Tools 1-6
1.4 What This Guide Covers and Does Not Cover 1-6

2 The Sample Application

2.1 Requirements for the Sample Application 2-2
2.2 Screenshots of the Sample Application 2-3
2.3 Database Schema 2-7

3 Application Design

3.1 Design Goals 3-2
3.2 Chaining Pages 3-3
3.3 Using Model-View-Controller (MVC) 3-4
3.3.1 MVC Diagram 3-4
3.3.2 Controller 3-6
3.3.3 Model (Business Logic) 3-6
3.3.4 View 3-7

4 Implementing Business Logic

4.1 Objects Needed by the Application 4-2
4.2 Other Options Considered But Not Taken 4-3
4.2.1 Conditions that Favor Using EJBs 4-3
4.2.2 Conditions that Favor Using Servlets 4-3
4.2.3 Conditions that Favor Using Normal Java Objects 4-3
4.3 Controller 4-5
4.4 Action Handlers 4-8
4.5 Employee Data (Entity Bean) 4-9
4.5.1 Home Interface 4-10
4.5.2 Remote Interface 4-11
4.5.3 Persistence 4-12
4.5.4 Load Method 4-12
4.5.5 EmployeeModel Class 4-12
4.5.6 Data Access Object for Employee Bean 4-14
4.5.6.1 Interface 4-14
4.5.6.2 Implementation 4-14
4.5.6.3 Load Method 4-16
4.6 Benefit Data (Stateless Session Bean) 4-18
4.6.1 Home Interface 4-18
4.6.2 Remote Interface 4-19
4.6.3 Benefit Details 4-20
4.7 EmployeeManager (Stateless Session Bean) 4-21
4.7.1 Home Interface 4-22
4.7.2 Remote Interface 4-23
4.8 Utility Classes 4-25

5 Creating Presentation Pages

5.1 HTML Files 5-1
5.2 Servlets 5-2
5.2.1 Automatic Compilation of Servlets 5-2
5.2.2 Example 5-3
5.2.3 Example: Calling an EJB 5-3
5.3 JSPs 5-5
5.3.1 Tag Libraries 5-5

5.3.2 Minimal Coding in JSPs 5-5
5.3.3 Multiple Client Types 5-6

6 Interaction Between Clients and Business Logic Objects

6.1 Client Interface to Business Tier Objects 6-2
6.2 Query Employee Operation 6-3
6.2.1 High-Level Sequence 6-3
6.2.2 Querying the Database and Retrieving Data 6-4
6.2.3 findByPrimaryKey Method 6-6
6.2.4 Getting Benefit Data 6-7
6.3 Add and Remove Benefit Operations 6-9
6.4 Add Benefit Operation 6-10
6.4.1 High-Level Sequence of Events 6-10
6.4.2 Getting Benefits That the User Can Add 6-11
6.4.3 Updating the Database 6-12
6.5 Removing Benefit Operation 6-14
6.5.1 High-Level Sequence of Events 6-14
6.5.2 Getting Benefits That the User Can Remove 6-15
6.5.3 Updating the Database 6-16

7 Supporting Wireless Clients

7.1 Changes You Need To Make To Your Application 7-2
7.2 Presentation Data for Wireless Clients 7-3
7.2.1 Screens for the Wireless Application 7-3
7.2.2 Differences Between the Wireless and the Browser Application 7-5
7.3 Deciding Where to Put the Presentation Data for Wireless Clients 7-7
7.3.1 Determining the Origin of a Request 7-7
7.3.2 Combining Presentation Data in the Same JSP File 7-8
7.3.3 Separating Presentation Data into Separate Files 7-10
7.4 Header Information in JSP Files for Wireless Clients 7-12
7.4.1 Setting the XML Type 7-12
7.4.2 Setting the Content Type 7-12
7.5 Operation Details 7-13
7.5.1 Query Operation 7-13
7.5.2 queryEmployeeWireless.jsp 7-15

7.5.3 Add and Remove Benefits Operations 7-16
7.6 Accessing the Application 7-18
7.6.1 Using a Simulator 7-18
7.6.2 Using an Actual Wireless Client 7-18

8 Adding Web Cache to the Application

8.1 Choosing Which Pages to Cache 8-2
8.2 Analyzing the Application 8-3
8.2.1 Specifying the Pages to Cache 8-3
8.2.2 Invalidating Pages 8-4
8.2.3 Setting up Triggers on the Underlying Tables 8-7

9 Running in a Portal Framework

9.1 How Portal Processes Requests 9-2
9.2 Screenshots of the Application in a Portal 9-3
9.3 Changes You Need to Make to the Application 9-7
9.3.1 Set up a Provider and a Portal Page 9-7
9.3.2 Edit the Application 9-8
9.4 Update the Links Between Pages Within a Portlet 9-9
9.4.1 The parameterizeLink Method 9-9
9.4.2 The next_page Parameter 9-10
9.4.3 Linking to the ID Page 9-11
9.5 Use include instead of the forward Method 9-12
9.6 Protect Parameter Names 9-13
9.6.1 Retrieving Values 9-14
9.6.2 Setting Values 9-14
9.7 Make All Paths Absolute 9-15
9.7.1 <a> and <link> Tags 9-15
9.7.2 <form> Tag 9-15

10 Enhancements to the Application

10.1 Adding Security Features 10-1
10.2 Publishing the Application as a Web Service 10-1

A Configuration Files

A.1 server.xml A-1
A.2 default-web-site.xml A-2
A.3 data-sources.xml A-2

Send Us Your Comments

Oracle9*i* Application Server Application Developer's Guide, Release 2 (9.0.2)

Part No. A95101-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: iasdocs_us@oracle.com
- FAX: 650-506-7407 Attn: Oracle9*i* Application Server Documentation Manager
- Postal service:
  Oracle Corporation
  Oracle9*i* Application Server Documentation
  500 Oracle Parkway, M/S 2op3
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

Oracle9*i* Application Server Application Developer's Guide describes how to create modular, extensible, and maintainable J2EE applications. It highlights how to structure your applications so that you get the maximum benefits from the features in Oracle9*i* Application Server. You should use this guide along with the OC4J User's Guide.

This preface contains these topics:

- Intended Audience
- Documentation Accessibility
- Related Documentation
- Conventions

Intended Audience

Oracle9*i* Application Server Application Developer's Guide is intended for developers who perform the following tasks:

- Design and create J2EE (with EJB, JSP, and servlets) applications
- Enhance applications to support wireless clients
- Enable applications to run in a portal framework

To use this document, you need to be familiar with Java and have some exposure to J2EE technology.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle Corporation is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Related Documentation

For more information, see these Oracle resources:

- *Oracle9i Application Server Concepts*
- Oracle9*i* Application Server Documentation Library
- Oracle9*i* Application Server Platform-Specific Documentation on Oracle9*i* Application Server Disk 1

In North America, printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://otn.oracle.com/admin/account/membership.html`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://otn.oracle.com/docs/index.htm`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples
- Conventions for Microsoft Windows Operating Systems

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width) font`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.
`lowercase monospace (fixed-width) font`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true.` Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`

|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}`
		`[COMPRESS | NOCOMPRESS]`

. . .	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS `*`subquery`*`;`
	■ That you can repeat a portion of the code	`SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	

Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);`
		`acct    CONSTANT NUMBER(4) := 3;`

Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`*
		`DB_NAME = `*`database_name`*

Convention	Meaning	Example

`UPPERCASE`	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		 `SELECT * FROM USER_TABLES;`
		 `DROP TABLE hr.employees;`

`lowercase`	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	 Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		 `sqlplus hr/hr`
		 `CREATE USER mjones IDENTIFIED BY ty3MU9;`

Conventions for Microsoft Windows Operating Systems

The following table describes conventions for Microsoft Windows operating systems and provides examples of their use.

Convention	Meaning	Example
Choose Start >	How to start a program.	To start the Oracle Database Configuration Assistant, choose Start > Programs > Oracle - *HOME_NAME* > Configuration and Migration Tools > Database Configuration Assistant.
File and directory names	File and directory names are not case sensitive. The following special characters are not allowed: left angle bracket (<), right angle bracket (>), colon (:), double quotation marks ("), slash (/), pipe (|), and dash (-). The special character backslash (\) is treated as an element separator, even when it appears in quotes. If the file name begins with \\, then Windows assumes it uses the Universal Naming Convention.	`c:\winnt"\"system32` is the same as `C:\WINNT\SYSTEM32`

Convention	Meaning	Example
`C:\>`	Represents the Windows command prompt of the current hard disk drive. The escape character in a command prompt is the caret (^). Your prompt reflects the subdirectory in which you are working. Referred to as the *command prompt* in this manual.	`C:\oracle\oradata>`

	The backslash (\) special character is sometimes required as an escape character for the double quotation mark (") special character at the Windows command prompt. Parentheses and the single quotation mark (’) do not require an escape character. Refer to your Windows operating system documentation for more information on escape and special characters.	`C:\>exp scott/tiger TABLES=emp QUERY=\"WHERE job=’SALESMAN’ and sal<1600\"`
		 `C:\>imp SYSTEM/`*`password`*` FROMUSER=scott TABLES=(emp, dept)`

HOME_NAME	Represents the Oracle home name. The home name can be up to 16 alphanumeric characters. The only special character allowed in the home name is the underscore.	`C:\> net start Oracle`*`HOME_NAME`*`TNSListener`

Convention	Meaning	Example
ORACLE_HOME and *ORACLE_BASE*	In releases prior to Oracle8*i* release 8.1.3, when you installed Oracle components, all subdirectories were located under a top level *ORACLE_HOME* directory that by default used one of the following names: ▪ C:\orant for Windows NT ▪ C:\orawin95 for Windows 95 ▪ C:\orawin98 for Windows 98 This release complies with Optimal Flexible Architecture (OFA) guidelines. All subdirectories are not under a top level *ORACLE_HOME* directory. There is a top level directory called *ORACLE_BASE* that by default is C:\oracle. If you install Oracle9*i* release 1 (9.0.1) on a computer with no other Oracle software installed, then the default setting for the first Oracle home directory is C:\oracle\ora90. The Oracle home directory is located directly under *ORACLE_BASE*. All directory path examples in this guide follow OFA conventions. Refer to *Oracle9i Database Getting Starting for Windows* for additional information about OFA compliances and for information about installing Oracle products in non-OFA compliant directories.	Go to the *ORACLE_BASE*\ORACLE_HOME\rdbms\admin directory.

1 Creating Applications: Overview

When you create applications to be deployed on Oracle9*i* Application Server, you might think of different ways to implement a feature. This guide walks you through the design and implementation of a sample application, and in the process of doing so, it discusses the available options for each feature and the advantages and disadvantages of each option.

The resulting application is modular and extensible: you can easily add features, add different client types (including wireless devices), and change the implementation of a feature with minimal impact on other features.

The sample application used in this guide is called "Employee Benefit Application". It enables users to view data such as employee name, phone, email, and job ID. Users can also add or remove their benefit elections. The application retrieves and updates data in an Oracle database.

The sample application makes use of many different technologies, including JavaServer Pages, servlets, Enterprise JavaBeans™, JDBC, portals, wireless devices, web cache, web services, JNDI, and JAAS.

Contents of this chapter:

- Section 1.1, "Overview of Oracle9iAS"
- Section 1.2, "Development Steps"
- Section 1.3, "Development Tools"
- Section 1.4, "What This Guide Covers and Does Not Cover"

1.1 Overview of Oracle9*i*AS

The Oracle9*i*AS platform supports many technologies; as a result, you have many choices when you design and create your applications. The following sections describe some of these key technologies.

1.1.1 J2EE

The J2EE support includes:

- Enterprise JavaBeans, which enable your applications to use entity, session, and message-driven beans. EJB comes with an EJB container that provides services for you. Services include transaction, persistence, and lifecycle management.
- Servlets, which enable you to generate dynamic responses to web requests.
- JavaServer Pages (JSP), which enable you to mix Java and HTML to author web applications easily. JSPs also enable you to generate dynamic responses to web requests.

  Servlets and JSPs run within a "web container", which also provides services similar to those provided by the EJB container.
- Java Authentication and Authorization Service (JAAS), which enables you to authenticate users (that is, it ensures that users are who they claim to be) and authorizes users (that is, it checks that the user has access to an object before executing or returning the object).
- Java Message Service (JMS), which enables you to send and receive data and events asynchronously.
- Java Transaction API (JTA), which enables your applications to participate in distributed transactions and access transaction services from other components.
- J2EE Connector Architecture, which enables you to connect and perform operations on enterprise information systems.

For complete J2EE details (including specifications), see:

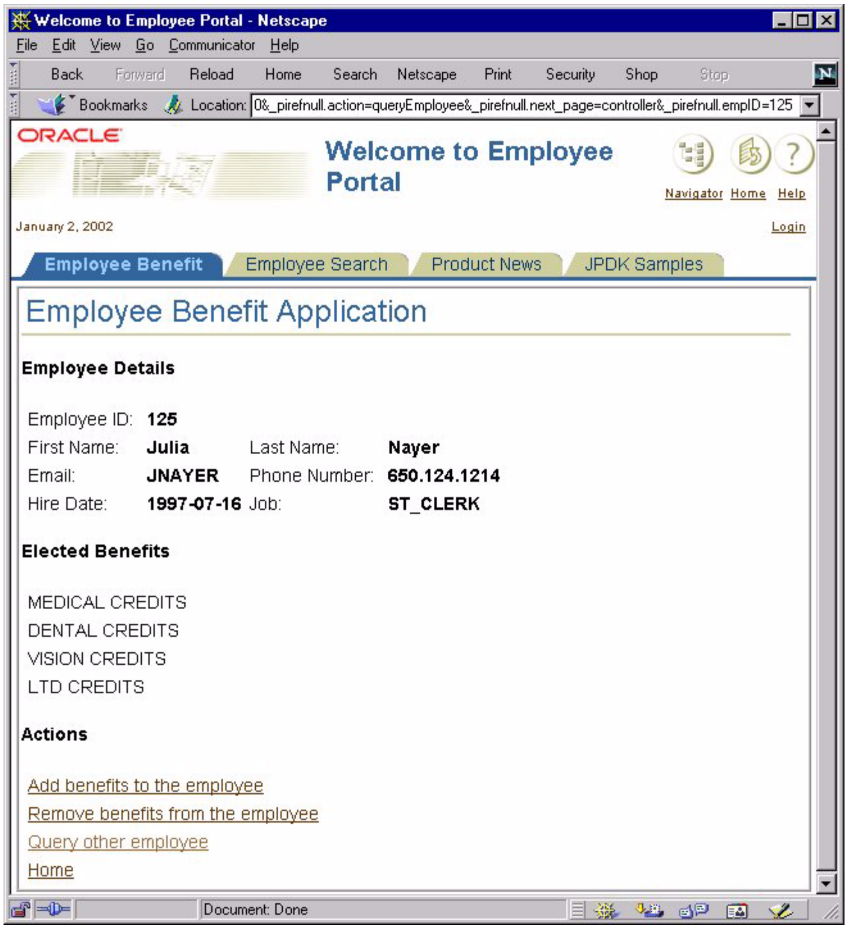
`http://java.sun.com/j2ee`

1.1.2 Enterprise Portals

Portals in Oracle9*i*AS enable you to aggregate, or group, your applications into a single web page. When users visit the page, they get a centralized location where they can see only the applications to which they have access with single sign-on capabilities. These applications, when displayed within a portal framework, are called portlets.

Figure 1–1 shows a picture of a portal.

Figure 1–1 A portal page


Welcome to Employee Portal - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: 0&_pirefnull.action=queryEmployee&_pirefnull.next_page=controller&_pirefnull.empID=125
ORACLE
Navigator Home Help
Login
Employee Benefit
Employee Search
Product News
JPDK Samples
Document: Done

Welcome to Employee Portal

January 2, 2002

Employee Benefit Application

Employee Details

Employee ID:	**125**		
First Name:	**Julia**	Last Name:	**Nayer**
Email:	**JNAYER**	Phone Number:	**650.124.1214**
Hire Date:	**1997-07-16**	Job:	**ST_CLERK**

Elected Benefits

MEDICAL CREDITS
DENTAL CREDITS
VISION CREDITS
LTD CREDITS

Actions

Add benefits to the employee
Remove benefits from the employee
Query other employee
Home

1.1.3 Wireless Support

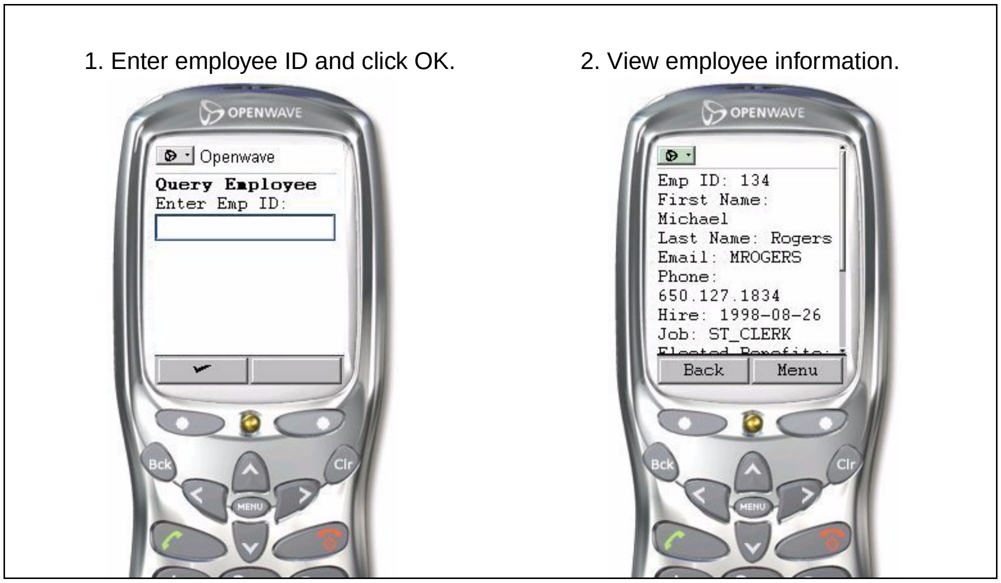
Browser clients do their rendering based on HTML tags, and there is more or less a standard set of tags and attributes that you can use. Wireless clients, on the other hand, understand different sets of tags and attributes, depending on the wireless device, and speak different protocols.

To make it easy for application developers, the wireless feature in Oracle9*i*AS comes with adaptors and transformers. This enables you to write your application once, and provide access to it from any wireless device. The way this works is that you write the presentation data in XML according to a standard DTD (document type definition), and the adaptors convert the XML on the fly to the markup language preferred by the client.

You can write your application such that it supports both browsers and wireless devices. Your application can check if a request is coming from a wireless client and return the appropriate response (HTML or XML). The sample application shows how to do that. See Chapter 7, "Supporting Wireless Clients" for details.

Figure 1–2 shows the Employee Benefit application running on a cell phone:

Figure 1–2 An application running on a cell phone



1. Enter employee ID and click OK.

```
Openwave
Query Employee
Enter Emp ID:
```

2. View employee information.

```
Emp ID: 134
First Name:
Michael
Last Name: Rogers
Email: MROGERS
Phone:
650.127.1834
Hire: 1998-08-26
Job: ST_CLERK
```

1.2 Development Steps

Designing and developing an application with all these technologies can be a little overwhelming. Here are some high-level steps to guide you (later chapters in this book provide the details):

1. Determine application requirements.

   Be sure to separate the presentation (or client) tier requirements from the business logic tier requirements. Separating the requirements by tier helps you design your application in a modular fashion. Modularity promotes a clean separation of functionality and enables you to reuse, update, or replace modules without affecting the rest of the application.

   See Section 2.1, "Requirements for the Sample Application" for details.

2. In the business logic tier, determine what objects you need and the interfaces of these objects.

   It helps to draw a sketch of the design based on the interfaces. Also determine how the client tier can invoke methods in the objects.

   When you determine what objects you need, you have many implementation choices. For example, you can use servlets, JavaBeans, Enterprise JavaBeans, or plain Java classes to implement your business logic.

   See Chapter 4, "Implementing Business Logic" for details.

3. In the client tier, create the presentation data for the client.

   The presentation data determine how the application looks to the users. Typically, the presentation data is in HTML (for browsers) or XML (for wireless devices). The HTML or XML tags can come from static files, JSPs, or other Java classes.

   JSPs and other Java classes can output the presentation data programmatically. In JSP files, you embed commands to invoke methods on the Java objects that implement your business logic. You can then display the values that the methods return.

   See Chapter 5, "Creating Presentation Pages" for details.

4. Implement the business logic.

   You can do this with EJBs, servlets, or other Java classes.

   See Chapter 4, "Implementing Business Logic" for details.

5. Package, deploy, and run your application.

1.3 Development Tools

To create applications for Oracle9*i*AS, you can use text editors such as emacs or vi, or you can use IDEs (integrated development environment).

If you use a text editor, you also need additional tools such as a Java compiler (for example, **javac**), a Java archive tool (for example, **jar**), and a packaging tool so that you can compile your files and build JAR and EAR files.

If you use IDEs, they can automate the tasks listed above for you. Oracle provides an IDE called Oracle9*i* JDeveloper. JDeveloper has support for each stage in the development lifecycle: it contains UML modelling and generation tools, debugging tools, profiling tools, and tuning tools.

JDeveloper is closely integrated with Oracle9*i*AS: you can deploy applications on Oracle9*i*AS from JDeveloper.

1.4 What This Guide Covers and Does Not Cover

This guide shows a complete application, clients (browsers and wireless devices), and database schema. It describes the logic behind the application design.

It also shows how to deploy/configure the application.

It does not describe the details for the APIs that the application uses. For that information, refer to the *Oracle9i*AS *J2EE Users Guide*.

This guide assumes the reader has some concept of servlets, JSPs, portals, web services, wireless devices, and introductory knowledge of EJBs. If you need more information on these topics, see the Oracle9*i*AS library for a list of books.

To read the Java specifications, see:

`http://java.sun.com`

2 The Sample Application

This chapter describes the sample application, "Employee Benefit Application", used in this guide.

Contents of this chapter:

- Section 2.1, "Requirements for the Sample Application"
- Section 2.2, "Screenshots of the Sample Application"
- Section 2.3, "Database Schema"

2.1 Requirements for the Sample Application

The Employee Benefit application enables users to view employee information (such as first name, last name, email, and phone number), and add and remove benefits. A typical user of the application is an employee who manages benefits for other employees in a company.

The functional requirements for the sample application are:

- Display data from the **employees**, **employee_benefit_items**, and **benefits** tables on the Info page (Figure 2–1). The database schema for these tables is listed in Section 2–4, "Database schema".
- Enable the user to add benefits.
- Enable the user to remove benefits.

Miscellaneous:

- Application must be able to run within a portal.

Clients for the application:

- Web browsers
- Wireless clients (mobile phones and PDAs)

2.2 Screenshots of the Sample Application

When the user invokes the application, the first page prompts the user to enter an employee ID (Figure 2–1).

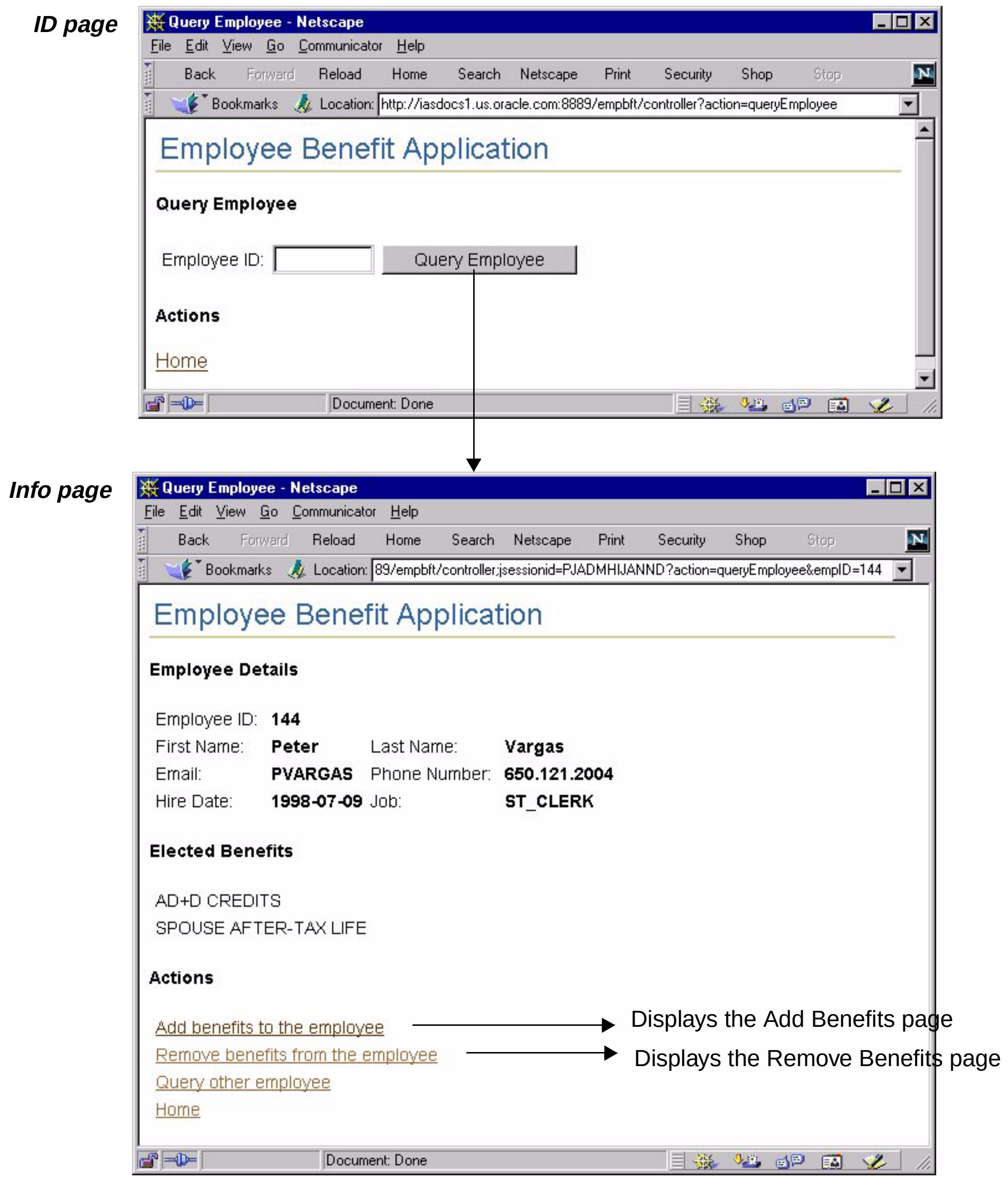
When the user clicks the Query Employee button, the application queries the database for the specified employee ID. If found, the application displays information for that employee, including which benefits the employee has currently elected (Figure 2–1).

If the employee ID does not match an employee, the application displays an error page (Figure 2–3).

On the Info page, the user can add or remove benefits by selecting the Add or Remove Benefit link. The application then displays the Add or Remove Benefits page (Figure 2–2). The user selects which benefits to add or remove, and clicks the Add Selected Benefits or Remove Selected Benefits button. If successful, the application displays the Success page, and the user can click the "Query the Same Employee" link to see the updated benefits.

For screen shots of the application running on a wireless device, see Chapter 7, "Supporting Wireless Clients".

Figure 2–1   ID page and Info page


ID page
Query Employee - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location:
Document: Done
Info page
Query Employee - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location:
Document: Done
Displays the Add Benefits page
Displays the Remove Benefits page

http://iasdocs1.us.oracle.com:8889/empbft/controller?action=queryEmployee

Employee Benefit Application

Query Employee

Employee ID: [ ] Query Employee

Actions

Home

89/empbft/controller;jsessionid=PJADMHIJANND?action=queryEmployee&empID=144

Employee Benefit Application

Employee Details

Employee ID:	**144**		
First Name:	**Peter**	Last Name:	**Vargas**
Email:	**PVARGAS**	Phone Number:	**650.121.2004**
Hire Date:	**1998-07-09**	Job:	**ST_CLERK**

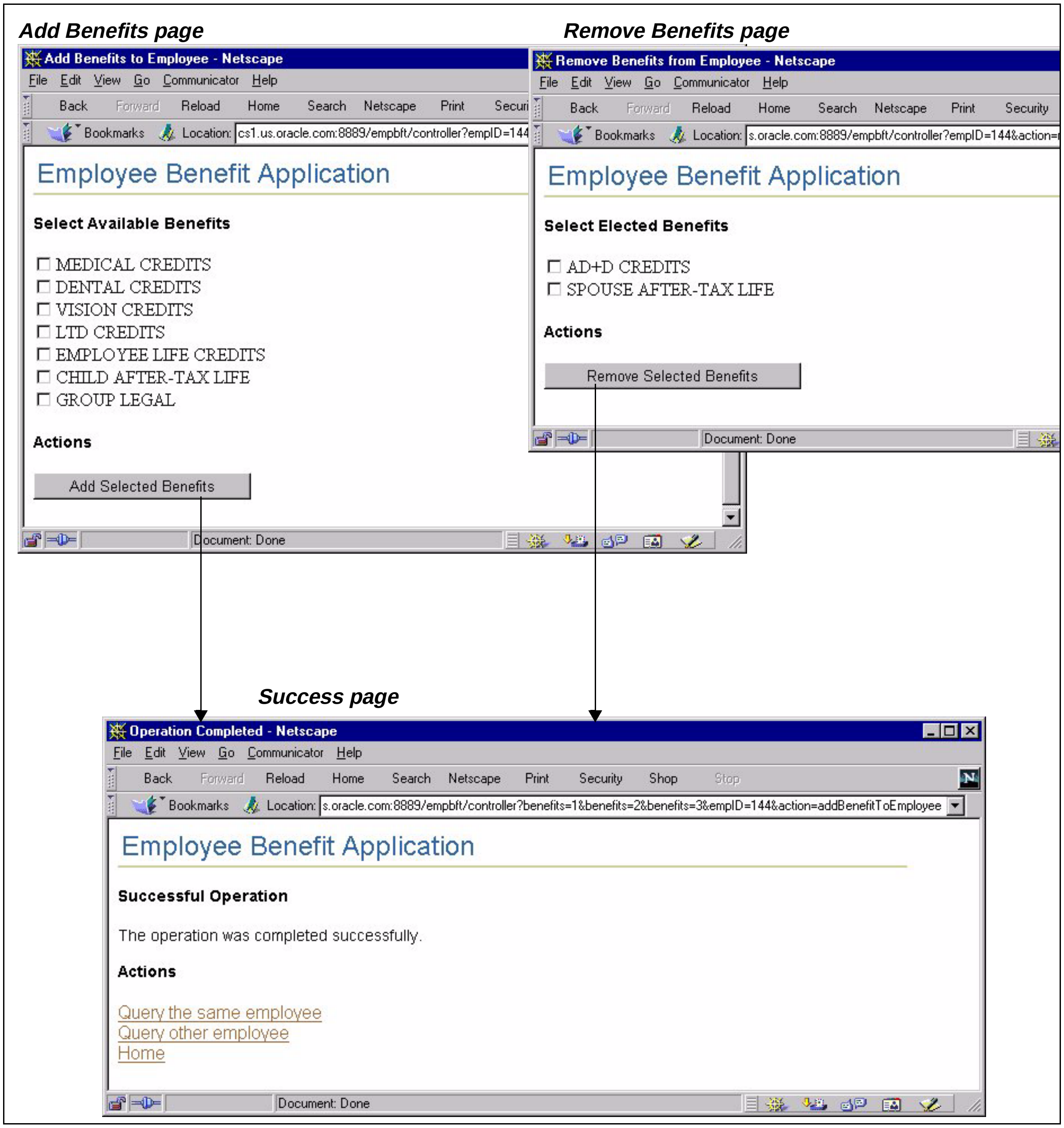
Elected Benefits

AD+D CREDITS
SPOUSE AFTER-TAX LIFE

Actions

Add benefits to the employee
Remove benefits from the employee
Query other employee
Home

Figure 2–2 Add Benefits Page, Remove Benefits Page, and Success Page



Add Benefits page

Add Benefits to Employee - Netscape

Location: cs1.us.oracle.com:8889/empbft/controller?empID=144

Employee Benefit Application

Select Available Benefits

☐ MEDICAL CREDITS
☐ DENTAL CREDITS
☐ VISION CREDITS
☐ LTD CREDITS
☐ EMPLOYEE LIFE CREDITS
☐ CHILD AFTER-TAX LIFE
☐ GROUP LEGAL

Actions

Add Selected Benefits

Document: Done

Remove Benefits page

Remove Benefits from Employee - Netscape

Location: s.oracle.com:8889/empbft/controller?empID=144&action=

Employee Benefit Application

Select Elected Benefits

☐ AD+D CREDITS
☐ SPOUSE AFTER-TAX LIFE

Actions

Remove Selected Benefits

Document: Done

Success page

Operation Completed - Netscape

Location: s.oracle.com:8889/empbft/controller?benefits=1&benefits=2&benefits=3&empID=144&action=addBenefitToEmployee

Employee Benefit Application

Successful Operation

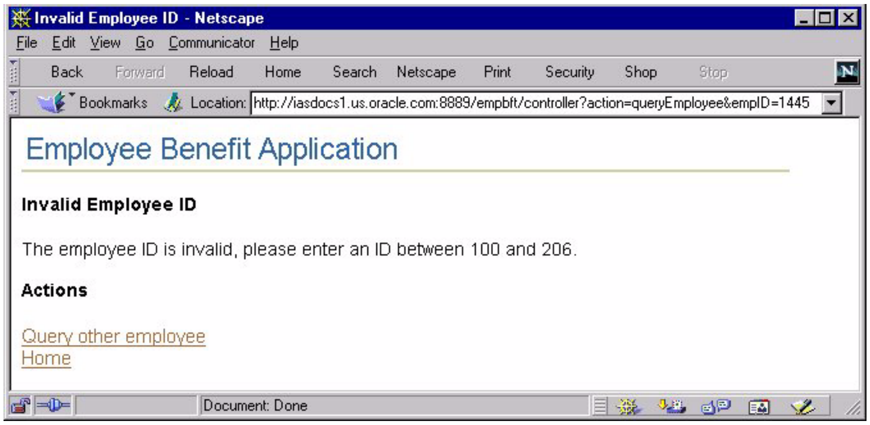
The operation was completed successfully.

Actions

Query the same employee
Query other employee
Home

Document: Done

Figure 2–3 Error page


Invalid Employee ID - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://iasdocs1.us.oracle.com:8889/empbft/controller?action=queryEmployee&empID=1445
Document: Done

Employee Benefit Application

Invalid Employee ID

The employee ID is invalid, please enter an ID between 100 and 206.

Actions

Query other employee
Home

2.3 Database Schema

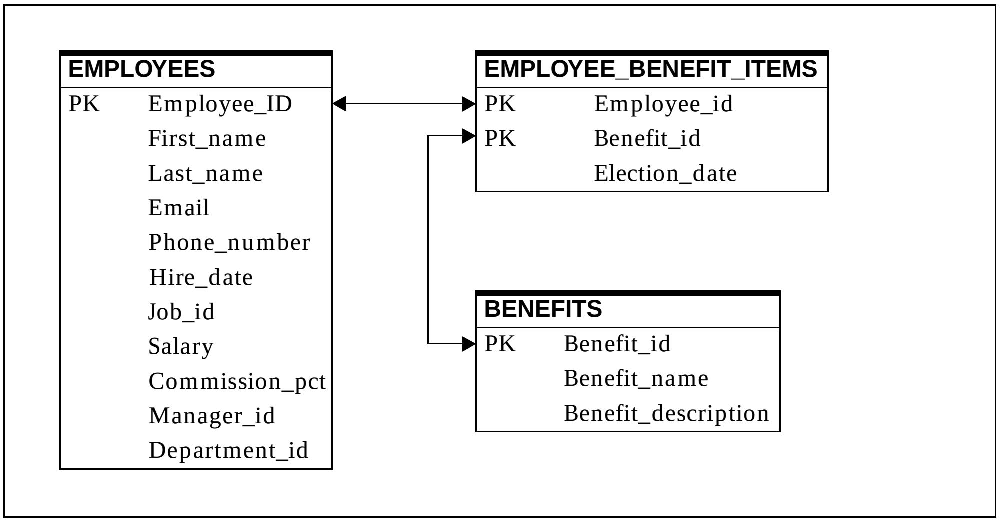
The Employee Benefit sample application uses the common **hr** schema that comes with Oracle9*i* database and Oracle9*i*AS Metadata Repository. The application uses the **hr.employees** table, plus two additional tables (**benefits** and **employee_benefit_items**) that you install. You install these tables in the default tablespace of the **hr** schema.

Table 2–1 Tables in the hr schema

Table name	Description
employees	Contains fields such as: employee_id, first_name, last_name, phone, email, and department.
benefits	Contains fields such as benefit_id, benefit_name, and benefit_description.
employee_benefit_items	Maps employees with benefits. The table has fields such as employee_id, employee_id, and election_date. An employee can have multiple benefits. This is the table that the application updates when employees update their benefit elections.

The application retrieves data from all three tables, but updates only the **employee_benefit_items** table.

Figure 2–4 Database schema



3 Application Design

There are several ways to design the architecture of the application described in Chapter 2, "The Sample Application". One way is to "chain" the pages, where page 1 calls page 2, page 2 calls page3, and so on. Another way is to use the model-view-controller (MVC) design pattern.

Contents of this chapter:

- Section 3.1, "Design Goals"
- Section 3.2, "Chaining Pages"
- Section 3.3, "Using Model-View-Controller (MVC)"

3.1 Design Goals

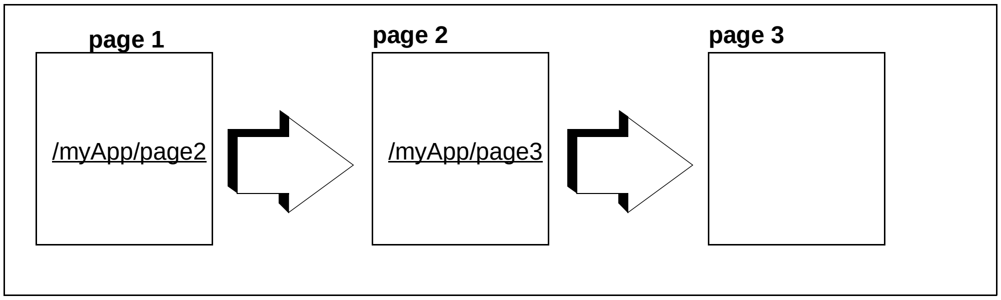
You want to design your application such that changes to one part of the application has minimal or no impact on other parts. This enables you to:

- Add features without redesigning your application
- Add new client types (such as wireless devices)
- Change client interfaces with minimal impact to your business logic
- Change business logic without changing presentation data
- Change your database schema or data source with minimal impact on your application

3.2 Chaining Pages

In the chaining pages design, pages in the application are linked sequentially. Page 1 has a link that calls page 2, page 2 has a link that calls page 3, and so on. Graphically:

Figure 3–1 Chaining pages



Each page can be generated differently. For example, the page 1 can be a plain HTML file, page 2 can be generated by a servlet, while page 3 can be generated by a JSP. The pages contain links or form elements (if the user needs to enter some values) to enable the user to get to the next page. In any case, the link to the next page is hardcoded on each page. See Chapter 5, "Creating Presentation Pages" for a discussion of generating HTML or other markup language.

Advantages of this design are that it is straightforward and easy to understand. This design is manageable for small applications that are unlikely to get bigger or whose pages are unlikely to change.

Disadvantages of this design are that there is no central point to handle client requests and it is difficult to move pages around. If pages get moved, added, or removed from the application, the application becomes less organized because you have to track down the code that one page calls and move it to another page, or change dependencies so that a page can be called from a different page.

3.3 Using Model-View-Controller (MVC)

A better way of designing this application is to use the MVC (model-view-controller) design pattern. MVC enables the application to be extensible and modular by separating the application into three parts:

- the business logic part, which implements data retrieval and manipulation
- the user interface part, which is what the application users see
- the controller part, which routes requests to the proper objects.

By separating an application into these parts, the MVC design pattern enables you to modify one part of the application without disturbing the other parts. This means that you can have multiple developers working on different parts of the application at the same time without getting into each other's domain. Each developer knows the role that each part plays in the application. For example, the user interface part cannot contain any code that has to do with business logic, and vice versa.

MVC also makes it easy to transform the application into a portlet or to have wireless devices access the application.

For more details on MVC, see:

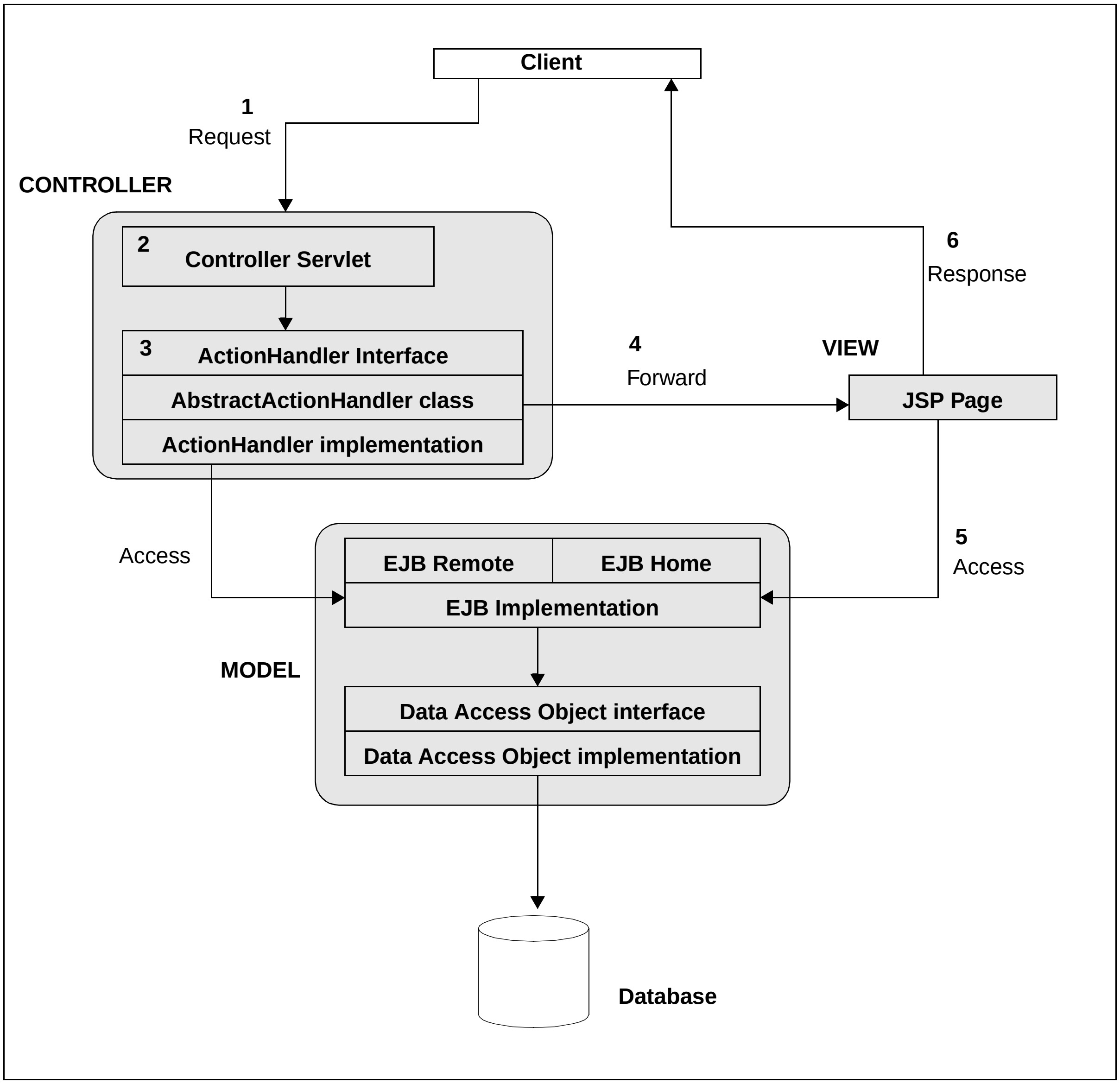
`http://java.sun.com/j2ee/blueprints/design_patterns/model_view_controller/index.html`

3.3.1 MVC Diagram

The following figure shows a high-level structure of the sample application. When the application receives a request from a client, it processes the request in the following manner:

1. The client sends a request, which is handled by the controller.
2. The controller determines the action specified by the request, and looks up the class for the action. The class must be a subclass of the AbstractActionHandler class.
3. The controller creates an instance of the class and invokes a method on that instance.
4. The instance processes the request. Typically, it forwards the request to a JSP page.
5. The JSP page gets an instance of the Enterprise JavaBean appropriate for the action and invokes the method to perform the action.
6. The JSP page then extracts the data that the method returned for presentation.

Figure 3–2 Application architecture



3.3.2 Controller

The controller is the first object in the application that receives requests from clients. All requests for any page in the application must first go through the controller.

In the controller, you map each request type with a class to handle the request. For example, the sample application has the following mappings:

Table 3–1 Mappings in the controller for the sample application

Action	Class
queryEmployee	empbft.mvc.handler.QueryEmployee
addBenefitToEmployee	empbft.mvc.handler.AddBenefitToEmployee
removeBenefitFromEmployee	empbft.mvc.handler.RemoveBenefitFromEmployee

The action is a query string parameter passed to the controller. The controller gets the value of the `action` parameter to determine the type.

When the controller receives a request, it looks up the value of the `action` parameter, determines the class for the request, creates an instance of the class, and sends the request to that instance.

You can hardcode the mapping in the controller code itself, or you can set up the controller to read the mapping information from a database or XML file. It is more flexible to use a database or XML file.

By having a controller as the first point of contact in your application, you can add functionality to your application easily. You just need to add a mapping and write the new classes to implement the new functionality.

In the sample application, the controller object is a servlet. The pages in the application have links to this servlet.

Using the controller object frees you from "chaining" pages in your application, where you have to keep track of which page calls which other pages, and adding or removing pages can be a non-trivial task.

3.3.3 Model (Business Logic)

The model represents the objects that implement your business logic. The objects process client data and return a response. The model also includes data from the database. Objects in the model can include Enterprise JavaBeans, JavaBeans, and regular Java classes. Views and controllers invoke objects in the model.

After the controller has read the request, objects in the model perform all the actual work in processing the request. For example, the objects can extract values from the query string and validate the request, authenticate the client, begin a transaction, and query databases. In the sample application, the **EmployeeManager** session bean calls the **Employee** entity bean to query the database and get information for an employee.

Although it is tempting to encode presentation data in your business logic, it is a better practice to separate the presentation data into its own file. For example, if you write the presentation data in a JSP file, you can edit the HTML markup in the file or change the format of the data without changing the model code. The page can then format the data accordingly. JSP files do not care how the methods get their data.

Typically, objects in the model read and update data in databases. If your application accesses databases, consider using DAO (data access objects) to separate the database access portion of your application from the rest of the model. This enables you to isolate the SQL statements that you send to the database.

Using DAOs gives you the flexibility to change your data source. If you update your database (for example, if you rename tables in the database or change the structure of tables in the database), you can update your SQL statements in the data access objects without changing the rest of your application.

The sample application uses a DAO to connect to the database. The DAO sets up a connection to the database, executes the required SQL statements on the database, and returns the data.

For additional information on how to create a "clean" model, you might want to read the J2EE blueprints page and the Design Patterns Catalog page on the Sun site:

```
http://java.sun.com/j2ee/blueprints
http://java.sun.com/j2ee/blueprints/design_patterns/catalog.html
```

3.3.4 View

The view includes presentation data such as HTML tags along with business data returned by the model. The presentation data and the business data are sent to the client in response to a request. The HTML tags usually have data and form elements (such as text fields and buttons) that the user can interact with, as well as other presentation elements.

The presentation data and the business data should come from different sources. The business data should come from the model, and the presentation data should come from JSP files. This way, you have a separation between presentation and business data.

One benefit of coding the business and presentation data separately is that it makes it easy to extend the application to support different client types. For example, you might need to extend your application to support wireless devices. Wireless devices read WML or other markup language, depending on the device. If you embed your presentation data in your business logic, it would be difficult to track which tag is for which client type. With the separation, you can reuse the same business objects with new presentation data.

In addition, new clients of the application might not even be graphical at all. They might not be interested in getting display tags. They might only be interested in getting a result, which they can process however they like.

The files for the presentation data should not contain any business logic code, other than invoking objects on the model side of the application. This enables you to change the implementation of the business logic and database schema without modifying the client code.

In the sample application, client types include browsers, different types of wireless devices, non-web clients (such as other applications), and SOAP clients. You can add clients or change how the data is presented to the clients just by changing the "view." The data can be HTML, WML, or any other markup language.

In the sample application, all the presentation code is in JSP files. The JSP files call on EJBs and servlets to process requests.

4 Implementing Business Logic

Recall that the Employee Benefit sample application follows the MVC design pattern. This chapter discusses the model (M) and the controller (C) in the application. The view (V) is covered in Chapter 5, "Creating Presentation Pages".

The business logic for the Employee Benefit application consists of listing the employee information, adding benefits, and removing benefits (see Section 2.1, "Requirements for the Sample Application") for a specific employee.

The database schema for the application, which you might find useful to review, is shown in Section 2.3, "Database Schema".

- Section 4.1, "Objects Needed by the Application"
- Section 4.2, "Other Options Considered But Not Taken"
- Section 4.3, "Controller"
- Section 4.4, "Action Handlers"
- Section 4.5, "Employee Data (Entity Bean)"
- Section 4.6, "Benefit Data (Stateless Session Bean)"
- Section 4.7, "EmployeeManager (Stateless Session Bean)"
- Section 4.8, "Utility Classes"

4.1 Objects Needed by the Application

JSP pages contain presentation data and they also invoke business logic objects to perform certain operations (query employee information, add benefits, and remove benefits). These objects can be plain Java classes or EJB objects.

The Employee Benefit application uses EJBs because it might offer more functions to users in the future. The EJB container provides services that the application might need.

What EJB objects does the application need?

- An object to manage employee data

  The application needs to query the database and display the retrieved data. This can be an entity bean.

- An object to contain master benefit data

  The application uses this object to determine which benefits a user does not have.

- A session bean to manage the employee entity beans
- A data access object (DAO)

  DAOs are used to connect to the data source. The EJBs do not connect to the data source directly.

- A Controller and ActionHandler objects

  These objects are needed to implement the MVC design pattern for the application.

- Utility objects

  The application uses utility objects to perform specific tasks. It has a class to print debugging messages, and a class to define constants used by other classes in the application.

4.2 Other Options Considered But Not Taken

The application could have used plain Java classes to hold data and not used EJBs at all. But if the application grows and contains more features, it might be easier to use EJBs because it comes with a container that provides services such as persistence and transactions.

Another advantage of using EJB is that it is easier to find developers who are familiar with the EJB standard. It takes longer for developers to learn a "home-grown" proprietary system.

Here are some guidelines to help you choose among EJBs, servlets, and normal Java objects.

4.2.1 Conditions that Favor Using EJBs

Choose EJBs when:

- You need to model complex business logic.
- You need to model complex relationships between business objects.
- You need to access your component from different client types such as JSPs and servlets.
- You need J2EE services.

4.2.2 Conditions that Favor Using Servlets

Choose servlets when:

- You need to maintain state but do not require J2EE services (**HttpSession** object).
- You do not need to dedicate servlet instances to individual clients. In large deployments with thousands of concurrent users, maintaining one stateful session bean instance for each client may be a bottleneck. Servlets provide a lighter weight alternative.
- You need to temporarily store state of business process within a single HTTP request and the request involves multiple beans.

4.2.3 Conditions that Favor Using Normal Java Objects

Choose normal Java objects when:

- You do not need built-in web and EJB services such as transactions, security, persistence, resource pooling.
- You need the following features that are not allowed in EJBs:
  - accessing a local disk using the **java.io** package
  - creating threads
  - using the **synchronized** keyword
  - using the **java.awt** or **javax.swing** packages
  - listening to a socket or creating a socket server
  - modifying the socket factory
  - using native libraries (JNI)
  - reading or writing static variables

4.3 Controller

The **Controller** servlet is the first object that handles requests for the application. It contains a mapping of actions to classes, and all it does is route requests to the corresponding class.

The **init** method in the servlet defines the mappings. In this case, the application hardcodes the mappings in the file. It would be more flexible if the mapping information comes from a database or a file.

When the Controller gets a request, it runs the **doGet** or the **doPost** method. Both methods call the **process** method, which looks up the value of the **action** parameter and calls the corresponding class.

```
package empbft.mvc;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.HashMap;
import empbft.util.*;

/** MVC Controller servlet. Implements the controller
    in an Model View Control pattern.
*/
public class Controller extends HttpServlet
{
  /* Private static String here, not String creation out of the execution
     path and hence help to improve performance. */
  private static final String CONTENT_TYPE = "text/html";

  /** Hashtable of registered ActionHandler object. */
  private HashMap m_actionHandlers = new HashMap();

  /** ActionHandlerFactory, responsible for instantiating ActionHandlers.  */
  private ActionHandlerFactory m_ahf = ActionHandlerFactory.getInstance();

  /** Servlet Initialization method.
      @param - ServletConfig
      @throws - ServletException
  */

  public void init(ServletConfig config) throws ServletException
  {
    super.init(config);
  //Register ActionHandlers in Hashtable,  Action name, implementation String
  //This really ought to come from a configuration file or database etc....
  this.m_actionHandlers.put(SessionHelper.ACTION_QUERY_EMPLOYEE,
    "empbft.mvc.handler.QueryEmployee");
  this.m_actionHandlers.put(SessionHelper.ACTION_ADD_BENEFIT_TO_EMPLOYEE,
    "empbft.mvc.handler.AddBenefitToEmployee");
 this.m_actionHandlers.put(SessionHelper.ACTION_REMOVE_BENEFIT_FROM_EMPLOYEE,
    "empbft.mvc.handler.RemoveBenefitFromEmployee");
}

/** doGet. Handle an MVC request. This method expects a parameter "action"
    http://localhost/MVC/Controller?action=dosomething&
                                 aparam=data&anotherparam=moredata
    @param - HttpServletRequest request,
    @param - HttpServletResponse response,
*/
public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
{
  process(request, response);
}


/** doPost. Handle an MVC request. This method expects a parameter "action"
    http://localhost/MVC/Controller?action=dosomething&
                                 aparam=data&anotherparam=moredata
    @param - HttpServletRequest request,
    @param - HttpServletResponse response,
*/
public void doPost(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
{
  process(request, response);
}

private void process(HttpServletRequest request, HttpServletResponse response)
{
  try
  {
    //Get the action from the request parameter
    String l_action = request.getParameter(SessionHelper.ACTION_PARAMETER);

    //Find the implemenation for this action
    if (l_action == null) l_action = SessionHelper.ACTION_QUERY_EMPLOYEE;
    String l_actionImpl = (String) this.m_actionHandlers.get(l_action);
```

```
      if (l_actionImpl == null) {
        throw new Exception("Action not supported.");
      }
      ActionHandler l_handler = this.m_ahf.createActionHandler(l_actionImpl);
      l_handler.performAction(request,response);
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

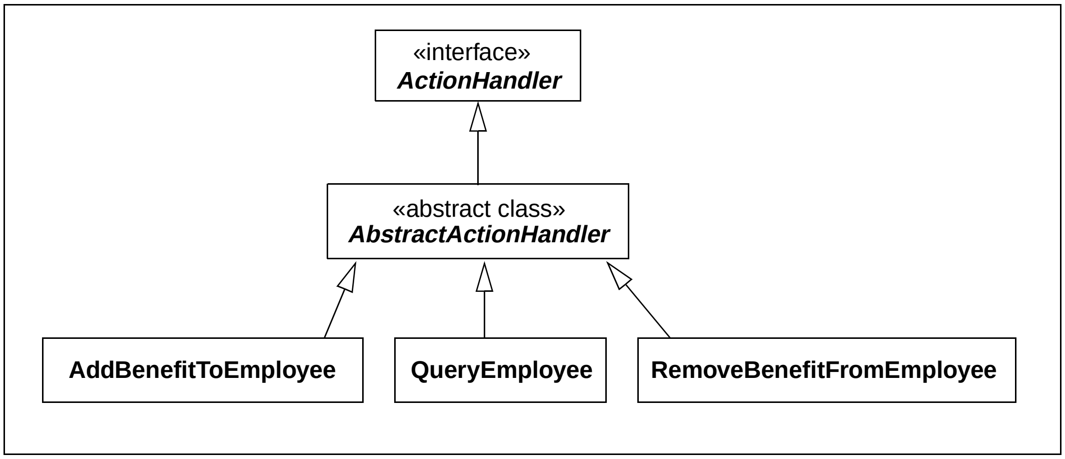
4.4 Action Handlers

The **process** method of the Controller servlet looks up the class that is mapped to the request and calls **createActionHandler** in the **ActionHandlerFactory** class to instantiate an instance of the mapped class.

The Employee Benefit application maps three actions to three classes. These classes must be subclasses of the **AbstractActionHandler** abstract class, which implements the **ActionHandler** interface, and must implement the **performAction** method. Figure 4–1 shows the relationship between these classes.

The **performAction** method checks whether the request came from a browser or a wireless device, and forwards the request to the appropriate JSP file. For browsers the JSP file returns HTML, while for wireless devices the JSP file returns XML. For example, the **performAction** method in **QueryEmployee.java** forwards requests from browsers to the **queryEmployee.jsp** file, but for requests from wireless devices the method forwards the requests to the **queryEmployeeWireless.jsp** file.

Figure 4–1 Action handlers



4.5 Employee Data (Entity Bean)

Employee data can be mapped to an **Employee** entity bean. The home and remote interfaces for the bean declare methods for retrieving employee data and adding and removing benefits.

Each instance of the bean represents data for one employee and the instances can be shared among clients. The EJB container instantiates entity beans and waits for requests to access the beans. By sharing bean instances and instantiating them before they are needed, the EJB container uses instances more efficiently and provides better performance. This is important for applications with a large number of clients.

Entity beans are less useful if the employees table is very large. The reason is that you are using a lot of fine-grained objects in your application.

Internally, the **Employee** bean stores employee data in a member variable called **m_emp** of type **EmployeeModel**. This class has methods for getting individual data items (such as email, job ID, phone).

Figure 4–2 Employee classes

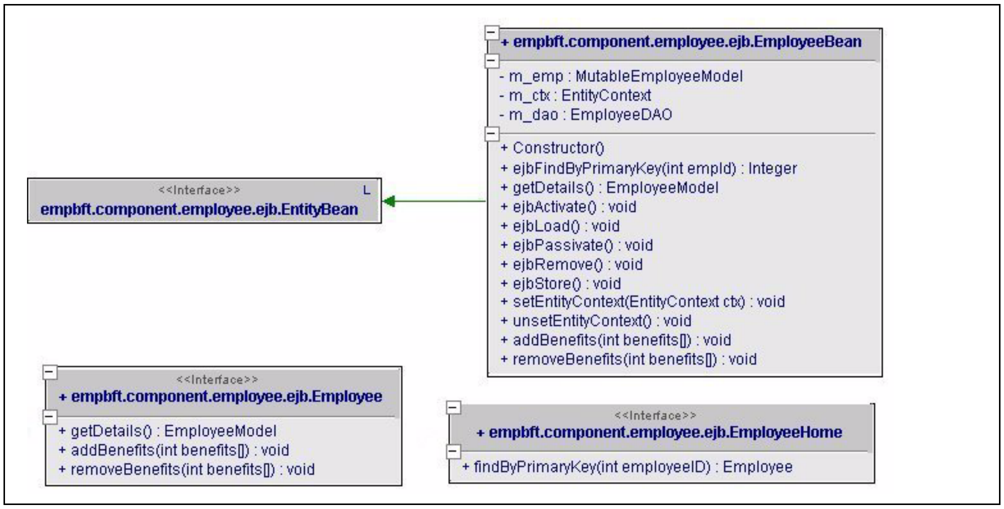
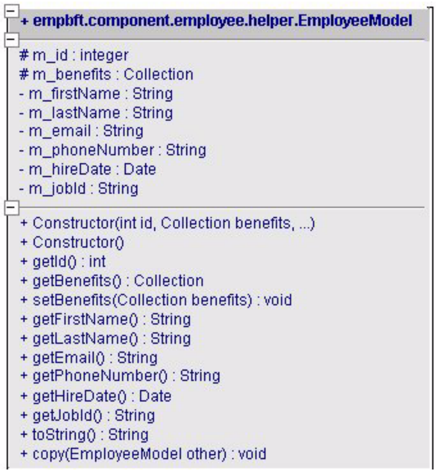

+ empbft.component.employee.ejb.EmployeeBean
- m_emp : MutableEmployeeModel
- m_ctx : EntityContext
- m_dao : EmployeeDAO
+ Constructor()
+ ejbFindByPrimaryKey(int empId) : Integer
+ getDetails() : EmployeeModel
+ ejbActivate() : void
+ ejbLoad() : void
+ ejbPassivate() : void
+ ejbRemove() : void
+ ejbStore() : void
+ setEntityContext(EntityContext ctx) : void
+ unsetEntityContext() : void
+ addBenefits(int benefits[]) : void
+ removeBenefits(int benefits[]) : void
<<Interface>>
empbft.component.employee.ejb.EntityBean
L
<<Interface>>
+ empbft.component.employee.ejb.Employee
+ getDetails() : EmployeeModel
+ addBenefits(int benefits[]) : void
+ removeBenefits(int benefits[]) : void
<<Interface>>
+ empbft.component.employee.ejb.EmployeeHome
+ findByPrimaryKey(int employeeID) : Employee

Figure 4–3   EmployeeModel class



4.5.1 Home Interface

The **Employee** entity bean has the following home interface:

```
package empbft.component.employee.ejb;
import java.rmi.RemoteException;
import javax.ejb.*;

public interface EmployeeHome extends EJBHome
{
  public Employee findByPrimaryKey(int employeeID)
    throws RemoteException, FinderException;
}
```

The **findByPrimaryKey** method, which is required for all entity beans, enables clients to find an **Employee** object. It takes an employee ID as its primary key.

It is implemented in the **EmployeeBean** class as **ejbFindByPrimaryKey**. To find an **Employee** object, it uses a data access object (DAO) to connect to the database and perform a query operation based on the employee ID.

See Section 4–6, "EmployeeDAO classes" for details on DAOs.

4.5.2 Remote Interface

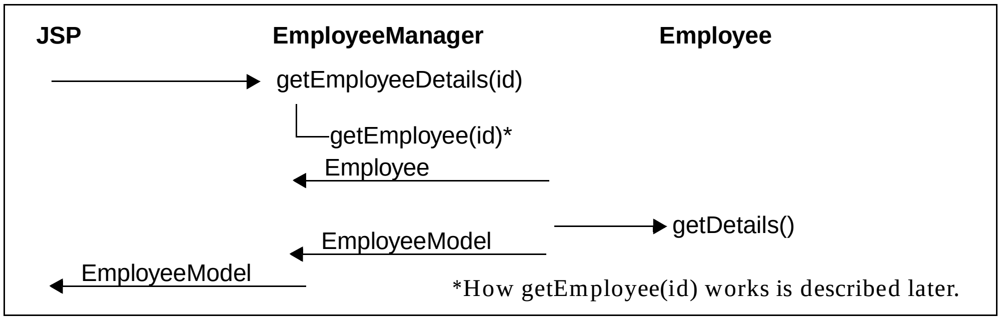
The **Employee** bean’s remote interface declares the methods for executing business logic operations:

```
package empbft.component.employee.ejb;
import java.rmi.RemoteException;
import javax.ejb.EJBObject;
import empbft.component.employee.helper.*;

public interface Employee extends EJBObject
{
    public EmployeeModel getDetails() throws RemoteException;
    public void addBenefits(int benefits[]) throws RemoteException;
    public void removeBenefits(int benefits[]) throws RemoteException;
}
```

The **addBenefits** and **removeBenefits** methods access the database using a DAO and perform the necessary operations. See Section 4.5.6, "Data Access Object for Employee Bean" for details.

The **getDetails** method returns an instance of **EmployeeModel**, which contains employee information. The query operation calls this method to get and display employee data. JSP pages call **getEmployeeDetails** method in **EmployeeManagerBean**, which call this method (Figure 4–4). Section 6.2, "Query Employee Operation" contains details on the query operation.

Figure 4–4 Getting employee details



4.5.3 Persistence

The **Employee** entity bean uses bean-managed persistence (BMP), rather than container-managed persistence. The bean controls when it updates data in the database.

It uses BMP because the employees-to-benefits is a many-to-many relationship, and an old version of Oracle9*i*AS (release 1022) does not support M:M relationship.

4.5.4 Load Method

The **Employee** entity bean implements the **ejbLoad** method, although the bean uses bean-managed persistence. The **ejbLoad** method queries the database (using the DAO) and updates the data in the bean with the new data from the database. This ensures that the bean's data is synchronized with the data in the database.

ejbLoad is called after the user adds or removes benefits.

```
// from EmployeeBean.java
public void ejbLoad() {
   try {
      if (m_dao == null)
          m_dao = new EmployeeDAOImpl();
      Integer id = (Integer)m_ctx.getPrimaryKey();
      this.m_emp = m_dao.load(id.intValue());
   } catch (Exception e) {
      throw new EJBException("\nException in loading employee.\n"
             + e.getMessage());
   }
}
```

See also Section 4.5.6.3, "Load Method", which describes the **load** method in the DAO.

4.5.5 EmployeeModel Class

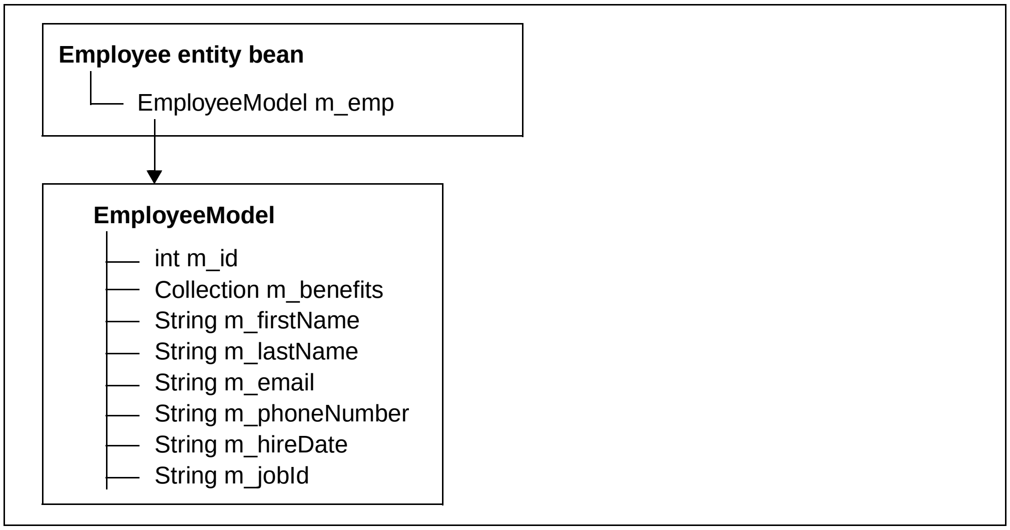
The implementation of the **Employee** bean uses a variable of type **EmployeeModel**, which contains all the employee details such as first name, last name, job ID, and so on. The following code snippet from **EmployeeBean** shows **m_emp** as a class variable:

```
public class EmployeeBean implements EntityBean
{
  private EmployeeModel m_emp;
  ...
}
```

Code snippet from **EmployeeModel**:

```
public class EmployeeModel implements java.io.Serializable
{
  protected int m_id;
  protected Collection m_benefits;
  private String m_firstName;
  private String m_lastName;
  private String m_email;
  private String m_phoneNumber;
  private Date m_hireDate;
  private String m_jobId;
...
}
```

Figure 4–5 Employee and EmployeeModel



4.5.6 Data Access Object for Employee Bean

Data access objects (DAO) are the only classes in the application that communicate with the database, or in general, with a data source. The entity and session beans in the application do not communicate with the data source. See Figure 3–2.

By de-coupling business logic from data access logic, you can change the data source easily and independently. For example, if the database schema or the database vendor changes, you only have to update the DAO.

DAOs have interfaces and implementations. EJBs access DAOs by invoking methods declared in the interface. The implementation contains code specific for a data source.

For details on DAOs, see:

http://java.sun.com/j2ee/blueprints/design_patterns/data_access_object/index.html

4.5.6.1 Interface

The **EmployeeDAO** interface declares the interface to the data source. Entity and session beans and other objects in the application call these methods to perform operations on the data source.

```
package empbft.component.employee.dao;
import empbft.component.employee.helper.EmployeeModel;

public interface EmployeeDAO {
  public EmployeeModel load(int id) throws Exception;
  public Integer findByPrimaryKey(int id) throws Exception;
  public void addBenefits(int empId, int benefits[]) throws Exception;
  public void removeBenefits(int empId, int benefits[]) throws Exception;
}
```

4.5.6.2 Implementation

The implementation of the DAO is called **EmployeeDAOImpl**. It uses JDBC to connect to the database and execute SQL statements on the database. If the data source changes, you need to update only the implementation, not the interface.

Employee and **Benefit** objects get an instance of the DAO and invoke the DAO's methods. The following example shows how the **addBenefits** method in the **Employee** bean invokes a method in the DAO.

```
// from EmployeeBean.java
```

```
public void addBenefits(int benefits[])
{
    try {
      if (m_dao == null) m_dao = new EmployeeDAOImpl();
      m_dao.addBenefits(m_emp.getId(), benefits);
      ejbLoad();
    } catch (Exception e) {
      throw new EJBException ("\nData access exception in adding benefits.\n"
        + e.getMessage());
    }
}
```

The **addBenefits** method in the **EmployeeDAOImpl** class looks like the following:

```
public void addBenefits(int empId, int benefits[]) throws Exception
{
   String queryStr = null;
   PreparedStatement stmt = null;
   try {
      getDBConnection();
      for (int i = 0; i < benefits.length; i ++) {
         queryStr = "INSERT INTO EMPLOYEE_BENEFIT_ITEMS "
                 + " (EMPLOYEE_ID, BENEFIT_ID, ELECTION_DATE) "
                 + " VALUES (" + empId + ", " + benefits[i] + ", SYSDATE)";
         stmt = dbConnection.prepareStatement(queryStr);
         int resultCount = stmt.executeUpdate();
         if (resultCount != 1) {
            throw new Exception("Insert result count error:" + resultCount);
         }
      }
   } catch (SQLException se) {
      throw new Exception(
          "\nSQL Exception while inserting employee benefits.\n"
          + se.getMessage());
   } finally {
       closeStatement(stmt);
       closeConnection();
   }
}
```

The methods in **EmployeeDAOImpl** use JDBC to access the database. Another implementation could use a different mechanism such as SQLJ to access the data source.

4.5.6.3 Load Method

After the **Employee** bean adds or removes benefits for an employee, it calls the **load** method in **EmployeeDAOImpl**:

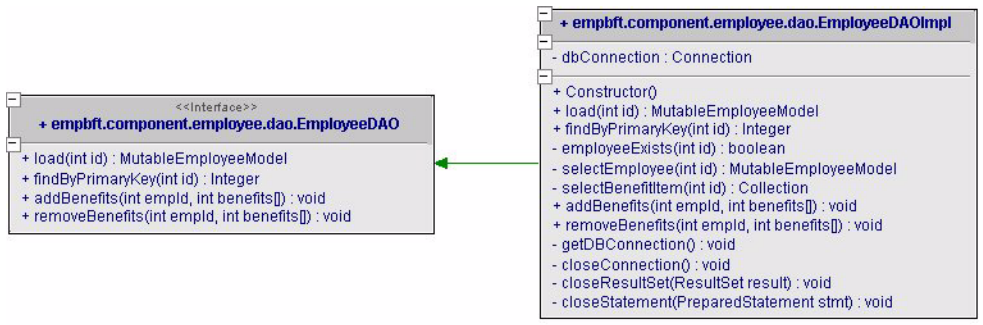
```
// from EmployeeBean.java
public void addBenefits(int benefits[])
{
   try {
      if (m_dao == null)
         m_dao = new EmployeeDAOImpl();
      m_dao.addBenefits(m_emp.getId(), benefits);
      ejbLoad();
   } catch (Exception e) {
      throw new EJBException ("\nData access exception in adding benefits.\n"
          + e.getMessage());
   }
}



// also from EmployeeBean.java
public void ejbLoad()
{
   try {
      if (m_dao == null)
         m_dao = new EmployeeDAOImpl();
      Integer id = (Integer) m_ctx.getPrimaryKey();
      this.m_emp = m_dao.load(id.intValue());
   } catch (Exception e) {
      throw new EJBException("\nException in loading employee.\n"
          + e.getMessage());
   }
}
```

The **ejbLoad** method in the **Employee** bean invokes **load** in the DAO object. By calling the **load** method after adding or removing benefits, the application ensures that the bean instance contains the same data as the database for the specified employee.

```
// from EmployeeDAOImpl.java
public EmployeeModel load(int id) throws Exception
{
   EmployeeModel details = selectEmployee(id);
   details.setBenefits(selectBenefitItem(id));
   return details;
}
```

Note that the EJB container calls **ejbLoad** in the **Employee** bean automatically after it runs the **findByPrimaryKey** method. See Section 6.2, "Query Employee Operation" for details.

Figure 4–6 EmployeeDAO classes


<<Interface>>
+ empbft.component.employee.dao.EmployeeDAO
+ load(int id) : MutableEmployeeModel
+ findByPrimaryKey(int id) : Integer
+ addBenefits(int empId, int benefits[]) : void
+ removeBenefits(int empId, int benefits[]) : void
+ empbft.component.employee.dao.EmployeeDAOImpl
- dbConnection : Connection
+ Constructor()
+ load(int id) : MutableEmployeeModel
+ findByPrimaryKey(int id) : Integer
- employeeExists(int id) : boolean
- selectEmployee(int id) : MutableEmployeeModel
- selectBenefitItem(int id) : Collection
+ addBenefits(int empId, int benefits[]) : void
+ removeBenefits(int empId, int benefits[]) : void
- getDBConnection() : void
- closeConnection() : void
- closeResultSet(ResultSet result) : void
- closeStatement(PreparedStatement stmt) : void

4.6 Benefit Data (Stateless Session Bean)

BenefitCatalog is a stateless session bean. It contains master benefit information such as benefit ID, benefit name, and benefit description for each benefit in the **benefits** table in the database.

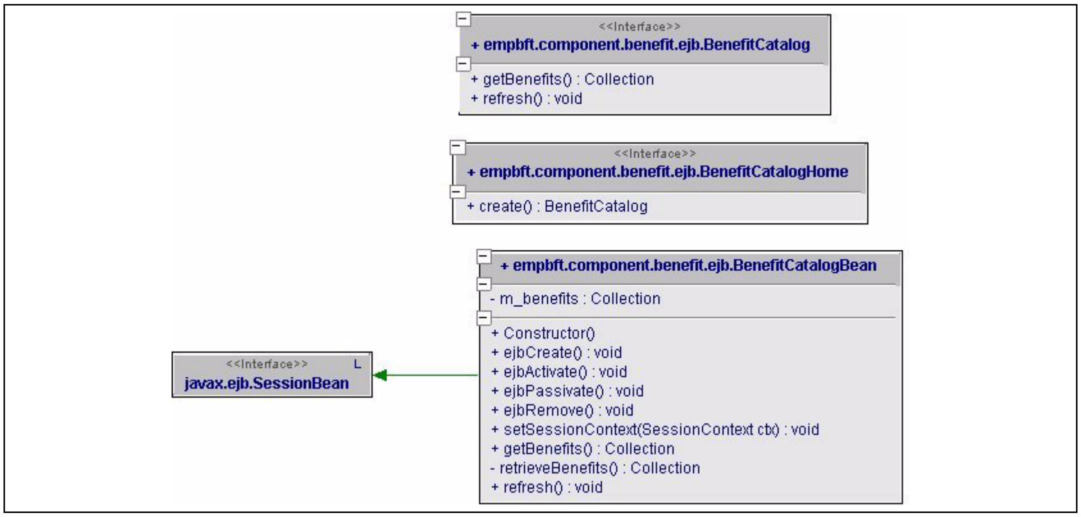
The application could have saved the benefit information to entity bean objects, but it uses a session bean instead. The reason for this is that the master benefit information does not change within the application. It is more efficient for a session bean to retrieve the data only once when the EJB container creates the bean.

Because the benefit information does not change, the **BenefitCatalog** bean does not need a data access object (DAO) to provide database access. The bean itself communicates with the database.

Each instance of the session bean contains all the benefit information. You can create and pool multiple instances for improved concurrency and scalability. If the application used entity beans and you mapped a benefit to a bean, it would have required one instance per benefit.

The bean is stateless so that one bean can be shared among many clients.

Figure 4–7 BenefitCatalog classes


<<Interface>>
+ empbft.component.benefit.ejb.BenefitCatalog
+ getBenefits() : Collection
+ refresh() : void
<<Interface>>
+ empbft.component.benefit.ejb.BenefitCatalogHome
+ create() : BenefitCatalog
+ empbft.component.benefit.ejb.BenefitCatalogBean
- m_benefits : Collection
+ Constructor()
+ ejbCreate() : void
+ ejbActivate() : void
+ ejbPassivate() : void
+ ejbRemove() : void
+ setSessionContext(SessionContext ctx) : void
+ getBenefits() : Collection
- retrieveBenefits() : Collection
+ refresh() : void
<<Interface>>
L
javax.ejb.SessionBean

4.6.1 Home Interface

The **BenefitCatalog** session bean has the following home interface:

```
package empbft.component.benefit.ejb;
import java.rmi.RemoteException;
import javax.ejb.EJBHome;
import javax.ejb.CreateException;

public interface BenefitCatalogHome extends EJBHome
{
  public BenefitCatalog create() throws RemoteException, CreateException;
}
```

The **create** method, which is implemented in **BenefitCatalogBean** as **ejbCreate**, queries the **benefits** table in the database to get a master list of benefits. The returned data (benefit ID, benefit name, benefit description) is saved to a **BenefitModel** object. Each record (that is, each benefit) is saved to one **BenefitModel** object.

The application gets a performance gain by retrieving the benefit data when the EJB container creates the bean, instead of when it needs the data. The application can then query the bean when it needs the data.

4.6.2 Remote Interface

The **BenefitCatalog** session bean has the following remote interface:

```
package empbft.component.benefit.ejb;
import java.rmi.RemoteException;
import javax.ejb.EJBObject;
import java.util.Collection;

public interface BenefitCatalog extends EJBObject
{
  public Collection getBenefits() throws RemoteException;
  public void refresh() throws RemoteException;
}
```

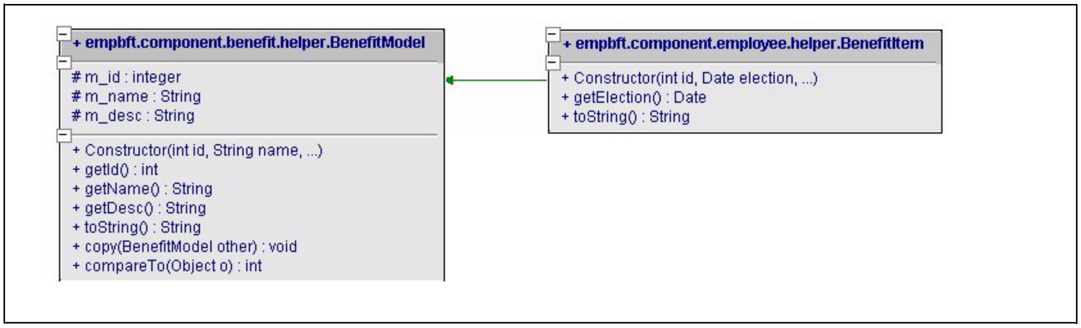
The **getBenefits** method returns a Collection of **BenefitModel**s. This is the master list of all benefits. This method is called by the **EmployeeManager** bean (by the **getUnelectedBenefitItems** method) when the application needs to display a user's unelected benefits. It compares a user's elected benefits against the master list, and displays the benefits that are not elected. The user then selects benefits to add from this list.

4.6.3 Benefit Details

The **BenefitCatalog** bean contains a Collection of **BenefitModel**s. The **BenefitModel** class contains the details (benefit ID, benefit name, and benefit description) for each benefit.

The **BenefitCatalog** bean contains a class variable called **m_benefits** of type Collection. Data in the Collection are of type **BenefitModel**. Each **BenefitModel** contains information about a benefit (such as benefit ID, name, and description). **BenefitItem** is a subclass of **BenefitModel**.

Figure 4–8 BenefitItem and BenefitModel classes


+ empbft.component.benefit.helper.BenefitModel
# m_id : integer
# m_name : String
# m_desc : String
+ Constructor(int id, String name, ...)
+ getId() : int
+ getName() : String
+ getDesc() : String
+ toString() : String
+ copy(BenefitModel other) : void
+ compareTo(Object o) : int
+ empbft.component.employee.helper.BenefitItem
+ Constructor(int id, Date election, ...)
+ getElection() : Date
+ toString() : String

JSPs call methods in **BenefitModel** to display benefit information. For example, **queryEmployee.jsp** calls the **getName** method to display benefit name.

```
<%
    Collection benefits = emp.getBenefits();
    if (benefits == null || benefits.size() == 0) {
  %>
      <tr><td>None</td></tr>
  <%
    } else {
      Iterator it = benefits.iterator();
      while (it.hasNext()) {
        BenefitItem item = (BenefitItem)it.next();
  %>
        <tr><td><%=item.getName()%></td></tr>
  <%
      } // end of while
    } // end of if
  %>
```

4.7 EmployeeManager (Stateless Session Bean)

EmployeeManager is a stateless session bean that manages access to the **Employee** entity bean. It is the only bean that JSPs can access directly; JSPs do not directly invoke the other beans (**Employee** and **BenefitCatalog**). To invoke methods on these beans, the JSPs go through **EmployeeManager**.

Generally, a JSP should not get an instance of an entity bean and invoke methods on the bean directly. It needs an intermediate bean that manages session state with clients and implements business logic that deals with multiple beans. Without this intermediate bean, you need to write the business logic on JSPs, and JSPs should have any business logic at all. A JSP's sole responsibility is to present data.

It is stateless because it does not contain data specific to a client.

EmployeeManager contains methods (defined in the remote interface) that JSPs can invoke to execute business logic operations. These methods invoke methods in the **Employee** and **BenefitCatalog** beans.

Table 4–1 Methods in EmployeeManager for business logic operations

Operation	Method
Query and display employee data	**getEmployeeDetails**(empID)
Add benefits	**getUnelectedBenefitItems**(empID)
Remove benefits	**getEmployeeDetails(empID)**, which returns **EmployeeModel**, then **getBenefits()** on the **EmployeeModel**

Examples:

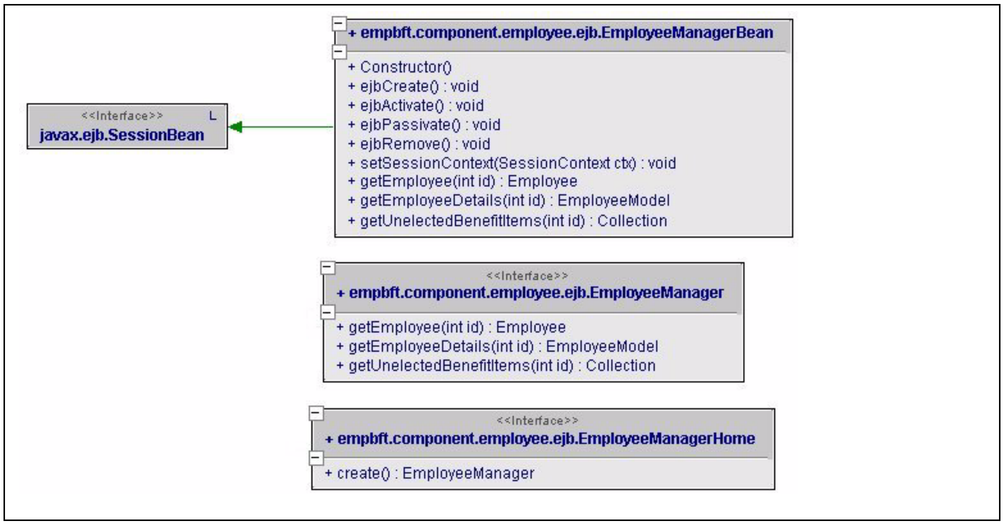
In **addBenefitToEmployee.jsp**:

```
<%
  int empId = Integer.parseInt(request.getParameter(
                               SessionHelper.EMP_ID_PARAMETER));
  EmployeeManager mgr = SessionHelper.getEmployeeManager(request);
  Collection unelected = mgr.getUnelectedBenefitItems(empId);
  ...
%>
```

In **removeBenefitFromEmployee.jsp**:

```
<%
  int empId = Integer.parseInt(request.getParameter(
                              SessionHelper.EMP_ID_PARAMETER));
  EmployeeManager mgr = SessionHelper.getEmployeeManager(request);
  Collection elected = mgr.getEmployeeDetails(empId).getBenefits();
  ...
%>
```

Figure 4–9 EmployeeManager classes



<<Interface>>
L
javax.ejb.SessionBean
+ empbft.component.employee.ejb.EmployeeManagerBean
+ Constructor()
+ ejbCreate() : void
+ ejbActivate() : void
+ ejbPassivate() : void
+ ejbRemove() : void
+ setSessionContext(SessionContext ctx) : void
+ getEmployee(int id) : Employee
+ getEmployeeDetails(int id) : EmployeeModel
+ getUnelectedBenefitItems(int id) : Collection
<<Interface>>
+ empbft.component.employee.ejb.EmployeeManager
+ getEmployee(int id) : Employee
+ getEmployeeDetails(int id) : EmployeeModel
+ getUnelectedBenefitItems(int id) : Collection
<<Interface>>
+ empbft.component.employee.ejb.EmployeeManagerHome
+ create() : EmployeeManager

4.7.1 Home Interface

The **EmployeeManager** has the following home interface:

```
package empbft.component.employee.ejb;

import java.rmi.RemoteException;
import javax.ejb.*;

public interface EmployeeManagerHome extends EJBHome
{
  public EmployeeManager create() throws RemoteException, CreateException;
}
```

The **create** method does nothing.

4.7.2 Remote Interface

The **EmployeeManager** has the following remote interface:

```
package empbft.component.employee.ejb;
import java.rmi.RemoteException;
import javax.ejb.EJBObject;
import java.util.Collection;
import empbft.component.employee.helper.*;

public interface EmployeeManager extends EJBObject
{
  public Employee getEmployee(int id) throws RemoteException;
  public EmployeeModel getEmployeeDetails(int id) throws RemoteException;
  public Collection getUnelectedBenefitItems(int id) throws RemoteException;
}
```

getUnelectedBenefitItems in **EmployeeManager** invokes methods on the **BenefitCatalog** bean and returns a Collection to the JSP, which iterates through and displays the contents of the Collection.

Methods in **EmployeeManager** also return non-bean objects to the application. For example, **queryEmployee.jsp** invokes the **getEmployeeDetails** method, which returns an **EmployeeModel**. The JSP can then invoke methods in **EmployeeModel** to extract the employee data.

```
// from queryEmployee.jsp
<%
   int id = Integer.parseInt(empId);
   EmployeeManager mgr = SessionHelper.getEmployeeManager(request);
   EmployeeModel emp = mgr.getEmployeeDetails(id);
   ...
%>
...
<table>
<tr><td>Employee ID: </td><td colspan=3><b><%=id%></b></td></tr>
<tr><td>First Name: </td><td><b><%=emp.getFirstName()%></b></td>
<td>Last Name: </td><td><b><%=emp.getLastName()%></b></td></tr>
```

Similarly, in **removeBenefitFromEmployee.jsp**, the page calls **getEmployeeDetails** to get an **EmployeeModel**, then it calls the **getBenefits** method on the **EmployeeModel** to list the benefits for the employee. The user can then select which benefits should be removed.

```
// from removeBenefitFromEmployee.jsp
<%
```

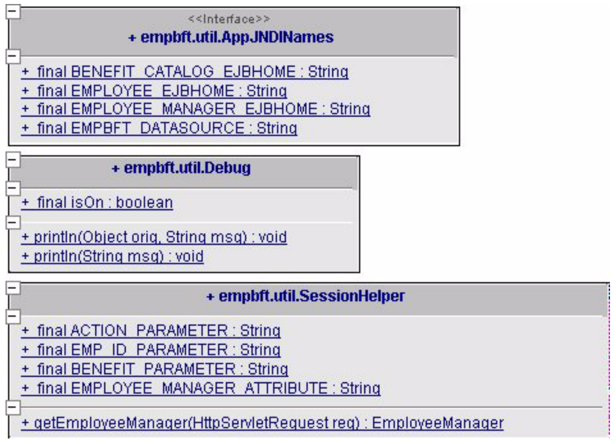
```
  int empId = Integer.parseInt(request.getParameter(
                        SessionHelper.EMP_ID_PARAMETER));
  EmployeeManager mgr = SessionHelper.getEmployeeManager(request);
  Collection elected = mgr.getEmployeeDetails(empId).getBenefits();
  ...
%>
...
<h4>Select Elected Benefits</h4>
<%
   Iterator i = elected.iterator();
   while (i.hasNext()) {
      BenefitItem b = (BenefitItem) i.next();
%>
<input type=checkbox name=benefits value=<%=b.getId()%>><%=b.getName()%><br>
<%
   }   // end while
%>
```

4.8 Utility Classes

The application uses these utility classes:

- **AppJNDINames** defines constants used to locate beans and classes.
- **Debug** contains methods that write messages to the window where you started OC4J.
- **SessionHelper** defines constants used to identify names of parameters in the query string.

Figure 4–10 Utility classes



<<Interface>>
+ empbft.util.AppJNDINames
+ final BENEFIT_CATALOG_EJBHOME : String
+ final EMPLOYEE_EJBHOME : String
+ final EMPLOYEE_MANAGER_EJBHOME : String
+ final EMPBFT_DATASOURCE : String
+ empbft.util.Debug
+ final isOn : boolean
+ println(Object orig, String msg) : void
+ println(String msg) : void
+ empbft.util.SessionHelper
+ final ACTION_PARAMETER : String
+ final EMP_ID_PARAMETER : String
+ final BENEFIT_PARAMETER : String
+ final EMPLOYEE_MANAGER_ATTRIBUTE : String
+ getEmployeeManager(HttpServletRequest req) : EmployeeManager

5 Creating Presentation Pages

You can create the presentation pages, which can contain data from business logic plus presentation elements, using different methods:

Contents of this chapter:

- Section 5.1, "HTML Files"
- Section 5.2, "Servlets"
- Section 5.3, "JSPs"

5.1 HTML Files

This option is valid for static pages only. If your pages have dynamic data, you have to generate the pages programmatically.

5.2 Servlets

Servlets enable you to generate pages programmatically. Using servlets, you can call business logic objects to obtain data, then assemble the page by adding in presentation elements. You can then send the completed page to the client.

Servlets can call methods in themselves and methods in other objects. Servlets can retrieve or update data in databases using JDBC or SQLJ.

Disadvantages of using servlets:

- Presentation elements are embedded with the business logic. This means that when you want to change the presentation code, you have to be careful not to change the business logic as well. In addition, the person editing the presentation code should have some knowledge of Java and not just HTML.
- Because presentation elements are embedded with the business logic, Oracle9*i*AS has to recompile the servlet when you change the presentation elements in the servlet.
- Another issue when using servlets to generate presentation elements is that you have to use the **println** method frequently. This makes the code look less tidy.

Servlets are a good choice for implementing state machines or controllers. State machines or controllers receive requests, make decisions based on parameters in the requests, and redirect the requests to the appropriate JSP for assembling the final display page to return to the clients. In the sample application, the controller is a servlet; see Section 4.3, "Controller".

See *Oracle9i*AS *Servlet Developer's Guide* for details on servlets.

5.2.1 Automatic Compilation of Servlets

One advantage to updating servlets is that Oracle9*i*AS has an auto-compile feature for servlets. You can place the uncompiled **.java** files for the servlets in the **$J2EE_ HOME/default-web-apps/WEB-INF/classes** directory, and Oracle9*i*AS will compile the files for you. To enable the auto-compile feature, set the **development** attribute of the **orion-web-app** tag to "true". This tag is found in **$J2EE_ HOME/home/config/global-web-application.xml**.

```
<orion-web-app
        jsp-cache-directory="./persistence"
        servlet-webdir="/servlet"
        development="true"
>
```

5.2.2 Example

For example, the following **doGet()** method in a servlet sends HTML data to the client:

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
                               throws ServletException, IOException {
    // Set the content type of the response
    res.setContentType("text/plain");

    // Get a print writer stream to write output to the response
    PrintWriter out = res.getWriter();

    // Send HTML to the output stream
    out.println("<HTML><HEAD>");
    out.println("<TITLE>Employee Benefit Application</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("<p>... more data here ...");
    // Close the HTML tags
    out.println("</BODY></HTML>");
}
```

5.2.3 Example: Calling an EJB

Here is an example of a servlet that calls an EJB object. Note how the servlet simply invokes methods on the EJB instance to get data. In this case, the servlet calls **getName()** and **getPrice()** methods on the EJB instance and embeds the return values within the presentation code.

```
import java.util.*;
import java.io.IOException;
import java.rmi.RemoteException;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

public class ProductServlet extends HttpServlet {
   ProductHome home;

   public void init() throws ServletException {
      try {
         Context context = new InitialContext();
```

```
            home = (ProductHome)PortableRemoteObject.
                            narrow(context.lookup("MyProduct"), ProductHome.class);
         }
         catch(NamingException e) {
            throw new ServletException("Error looking up home", e);
         }
      }

      public void doGet(HttpServletRequest request, HttpServletResponse response)
                        throws ServletException, IOException {
         response.setContentType("text/html");
         ServletOutputStream out = response.getOutputStream();
         try {
            Collection products = home.findAll();

            out.println("<html>");
            out.println("<head><title>My products</title></head>");
            out.println("<body>");
            out.println("<table border=\"2\">");
            out.println("<tr><td><b>Name</b></td><td><b>Price</b></td></tr>");

            Iterator iterator = products.iterator();
            while (iterator.hasNext())
            {
               Product product = (Product)PortableRemoteObject.
                                          narrow(iterator.next(), Product.class);
               out.println("<tr><td>" + product.getName() + "</td><td>" +
                               product.getPrice() + "</td></tr>");
            }
            out.println("</table>");
            out.println("</body>");
            out.println("</html>");
         }
         catch(RemoteException e) {
            out.println("Error communicating with EJB-server: " + e.getMessage());
         }
         catch(FinderException e) {
            out.println("Error finding products: " + e.getMessage());
         }
         finally {
            out.close();
         }     // finally
      }     // doGet method
   }
```

5.3 JSPs

Like servlets, JSP files enable you to combine HTML tags with Java commands. You do not have the **println** statements in JSP files like you do in servlets. Instead, you write your HTML tags as usual, but you add in special tags for JSP commands.

JSPs can do everything that servlets can do. For example, JSPs can invoke other classes and connect to the database to retrieve data or update data in the database.

See *Oracle9i*AS *Support for JavaServer Pages Reference* for details on JSPs.

5.3.1 Tag Libraries

In addition, JSPs enable you to define custom tags in tag libraries. Tag libraries enable you to define the behavior of your custom tags. Your JSPs can then access the tag libraries and use the custom tags. This enables you to standardize presentation and behavior across all your JSP files.

Here are few examples of how you can use custom tags and tag libraries. You can use them to:

- Send email. Tag libraries can hide the details of JavaMail API.
- Access web services.
- Access UltraSearch tags.
- Upload or download content from a file or database.

See *Oracle9i*AS *JSP Tag Libraries and Utilities Reference* for details on tag libraries.

5.3.2 Minimal Coding in JSPs

Although you can use as much Java in your JSPs as you like, the file can be difficult to read and debug if it is interleaved with JSP scriptlets and HTML. You will get a cleaner design for your application if you place all the business logic code outside the JSP files. The JSP scriptlets in your files can call out to Enterprise JavaBeans and other Java classes to run business logic. These objects then return the data or status to your JSP file, where you can extract the data and display the data using HTML or XML.

Another benefit of excluding business logic code from your JSPs is that you can have web page designers who might not be familiar with Java work on the JSP page. They can design the look of the page, using placeholders for the real data. Your developers, who might not want to bother with HTML, can be working on the business logic in other files simultaneously.

5.3.3 Multiple Client Types

If you are supporting different client types (browsers and wireless clients), you can have two versions of JSP files: one that returns HTML and one that returns XML. One important note is that both files make the same calls to the same objects to perform business logic. This is what the sample application does.

In the Employee Benefit application, all the presentation code, even the pages for error conditions, are written in JSP files, and the JSP files do not contain any business logic code. The application uses one file for browsers (for example, **addBenefitToEmployee.jsp**) and a similar file for wireless clients (for example, **addBenefitToEmployeeWireless.jsp**). The wireless version of the file contains XML instead of HTML.

6 Interaction Between Clients and Business Logic Objects

Previous chapters describe the JSP client files and the business logic objects. This chapter describes how these objects interact with each other: it shows how the JSP files access objects and retrieve data from the objects.

Contents of this chapter:

- Section 6.1, "Client Interface to Business Tier Objects"
- Section 6.2, "Query Employee Operation"
- Section 6.3, "Add and Remove Benefit Operations"
- Section 6.4, "Add Benefit Operation"
- Section 6.5, "Removing Benefit Operation"

6.1 Client Interface to Business Tier Objects

Although some methods in the business tier objects are declared public, client tier objects (that is, the JSP files) should access only some of these objects and methods. The methods are declared public so that other business tier objects can invoke them.

JSP files do not invoke methods on the **Employee** bean or the **BenefitCatalog** bean directly. Instead, the files invoke methods on an **EmployeeManager** bean, and these methods invoke methods on the **Employee** or **BenefitCatalog** objects. The **EmployeeManager** class has methods to execute the business logic operations. See **Section 4.7, "EmployeeManager (Stateless Session Bean)"** for details.

To get a reference to the **EmployeeManager** bean, the JSP files reference the **SessionHelper** class, which is a "regular" Java class. The **SessionHelper** class contains the **getEmployeeManager** static method which returns an instance of **EmployeeManager**. The **SessionHelper** class instantiates and stores the session bean in an attribute of **HttpSession** class. For example:

```
// from addBenefitToEmployee.jsp
<%
  int empId = Integer.parseInt(request.getParameter(
                              SessionHelper.EMP_ID_PARAMETER));
  EmployeeManager mgr = SessionHelper.getEmployeeManager(request);
  Collection unelected = mgr.getUnelectedBenefitItems(empId);
  ...
%>
```

6.2 Query Employee Operation

Typically, the user accesses the application through a link on an external page. The link's URL looks like this:

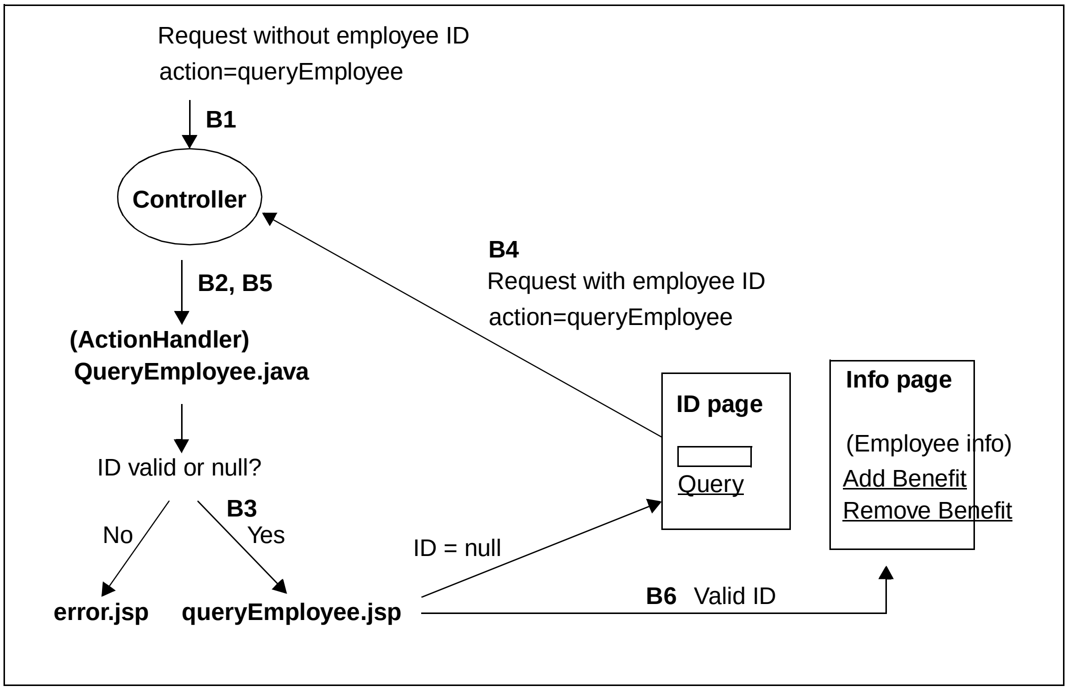
```
http://<host>/empbft/controller?action=queryEmployee
```

The user then sees the ID page (Figure 2–1).

6.2.1 High-Level Sequence

Figure 6–1 diagrams the query operation. The numbers in the figure correspond to the steps that follow the figure. This figure covers requests from browsers only. See Section 7.5.1, "Query Operation" for requests from wireless clients.

Figure 6–1 Query Operation



B1: The **Controller** servlet handles the request to the application.

B2: The value of the **action** parameter is `queryEmployee`, so the **Controller** invokes the **performAction** method in the **QueryEmployee** class.

B3: The **performAction** method forwards the request to the **queryEmployee.jsp** file, which displays an ID page (Figure 2–1).

B4: The user then enters an employee ID and clicks Query. The request still has the same value in the **action** parameter (`queryEmployee`), but it also has an employee ID parameter. The request is again handled by the **QueryEmployee** class.

B5: The **performAction** method in the **QueryEmployee** class and the **queryEmployee.jsp** file validate the employee ID entered by the user.

B6: For valid employee IDs, the JSP file queries the database to retrieve data for the specified employee ID.

6.2.2 Querying the Database and Retrieving Data

To get employee details, **queryEmployee.jsp** invokes the **getEmployeeDetails(employeeId)** method in **EmployeeManager**. The method returns an **EmployeeModel** object, which contains the data. The JSP then retrieves values from the **EmployeeModel** object to display the employee data.

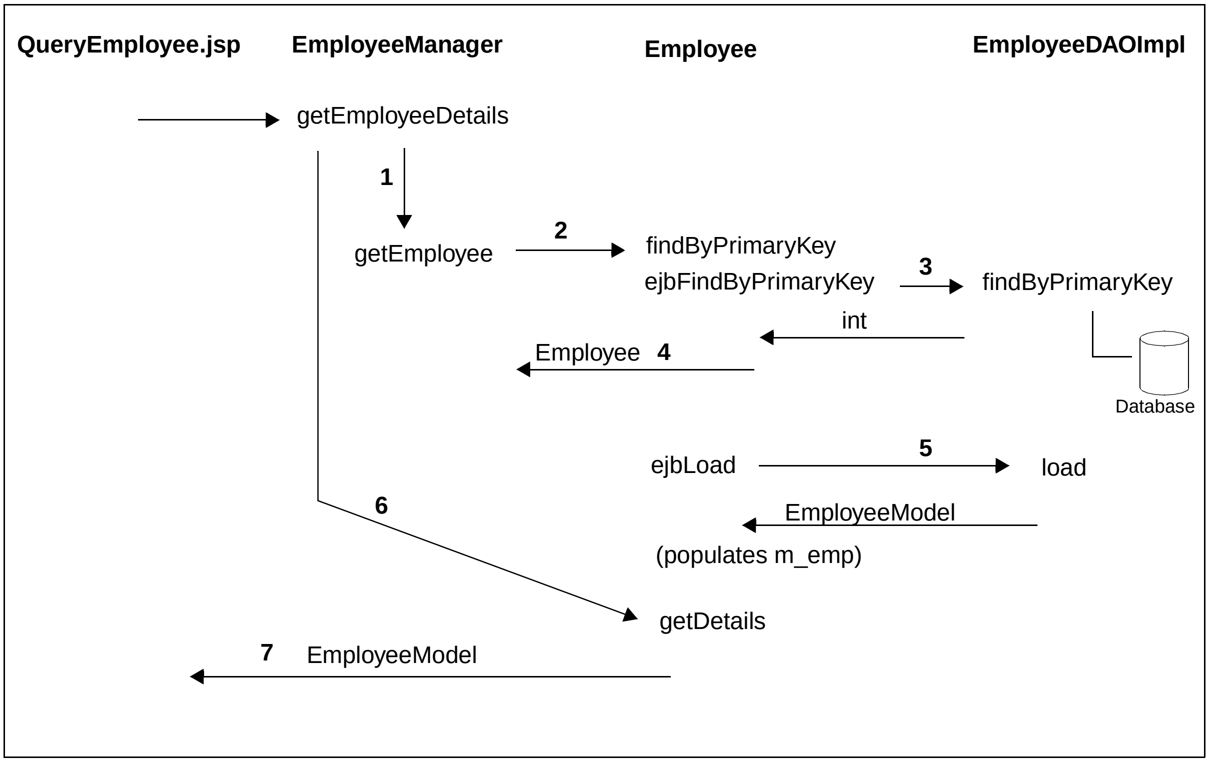
```
// from queryEmployee.jsp
<%
   ...
   int id = Integer.parseInt(empId);
   EmployeeManager mgr = SessionHelper.getEmployeeManager(request);
   EmployeeModel emp = mgr.getEmployeeDetails(id);
   ...
%>
...
<h4>Employee Details</h4>
<table>
<tr><td>Employee ID: </td><td colspan=3><b><%=id%></b></td></tr>
<tr><td>First Name: </td><td><b><%=emp.getFirstName()%></b></td><td>Last Name:
</td><td><b><%=emp.getLastName()%></b></td></tr>
<tr><td>Email: </td><td><b><%=emp.getEmail()%></b></td><td>Phone Number:
</td><td><b><%=emp.getPhoneNumber()%></b></td></tr>
<tr><td>Hire Date:
</td><td><b><%=emp.getHireDate().toString()%></b></td><td>Job:
</td><td><b><%=emp.getJobId()%></b></td></tr>
</table>
```

The **getEmployeeDetails** method in **EmployeeManager** starts off the following sequence:

1. It calls **getEmployee** to get an instance of the desired employee.

2. **getEmployee** invokes **findByPrimaryKey** on the **Employee** class. This calls **ejbFindByPrimaryKey** in **EmployeeBean**.
3. **ejbFindByPrimaryKey** calls **findByPrimaryKey** in **EmployeeDAOImpl**, which returns an int.
4. This int enables the EJB container to return an **Employee** bean from **findByPrimraryKey**, as declared in the **Employee** home interface.
5. Note that **findByPrimaryKey** in the **Employee** class is a special method. When you invoke this method, the EJB container automatically calls **ejbLoad** for you. **ejbLoad** calls **load** in **EmployeeDAOImpl**, which returns an **EmployeeModel**. This is used to populate the **m_emp** class variable.
6. **getEmployeeDetails** then calls **getDetails** with the **Employee** bean returned from step 1.
7. **getDetails** returns an **EmployeeModel** to the JSP.

Figure 6–2 getEmployeeDetails



6.2.3 findByPrimaryKey Method

The **EmployeeBean** class implements the **ejbFindByPrimaryKey(int empId)** method. This method calls the **EmployeeDAOImpl** class to retrieve data from the database.

```
// from EmployeeBean.java
public Integer ejbFindByPrimaryKey(int empId) throws FinderException
{
    try {
      if (m_dao == null) m_dao = new EmployeeDAOImpl();
      Integer findReturn = m_dao.findByPrimaryKey(empId);
      return findReturn;
    } catch (Exception e) {
      throw new FinderException ("\nSQL Exception in find by primary key.\n"
        + e.getMessage());
    }
}
```

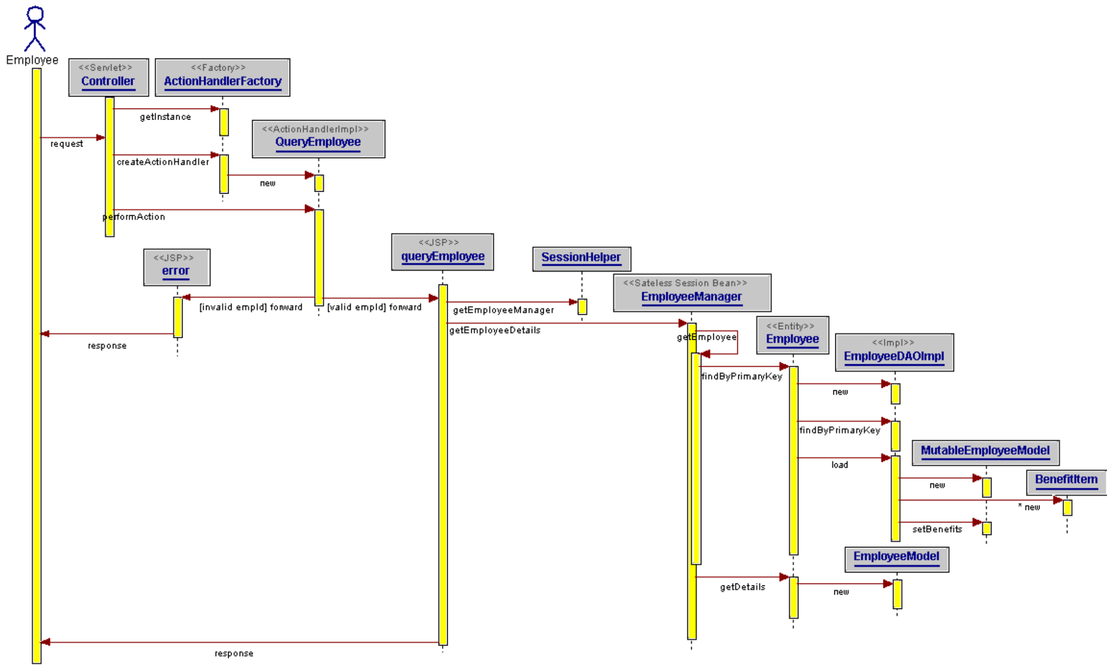
In the **EmployeeDAOImpl** class the **findByPrimaryKey(int id)** method queries the database for the specified employee ID. It executes a SELECT statement on the database and returns the employee ID if it finds an employee. If it does not find an employee, it throws an exception.

6.2.4 Getting Benefit Data

For benefit data, where a user can have more than one benefit, the application iterates over the Collection.

```
// from queryEmployee.jsp
<h4>Elected Benefits</h4>
<table>
  <%
    Collection benefits = emp.getBenefits();
    if (benefits == null || benefits.size() == 0) {
  %>
      <tr><td>None</td></tr>
  <%
    } else {
      Iterator it = benefits.iterator();
      while (it.hasNext()) {
        BenefitItem item = (BenefitItem)it.next();
  %>
        <tr><td><%=item.getName()%></td></tr>
  <%
      } // end of while
    } // end of if
  %>
</table>
```

Figure 6–3 Sequence Diagram for Query Employee



6.3 Add and Remove Benefit Operations

For the add and remove operations, the JSPs send which benefit to add or remove, plus the employee ID, to the **EmployeeManager**. The **EmployeeManager** adds or removes the benefit and returns the status of the operation.

The add and remove benefits operations follow similar sequences in presenting a list of benefits to the user, and executing the add or remove operation on the database.

- To add benefits, the user clicks the Add Benefit link on the Info page (Figure 2–1). The URL behind this link looks like:

  ```
  <a href="/empbft/controller?empID=<%=id%>&amp;action=addBenefitToEmployee">
  Add benefits to the employee</a>
  ```

- To remove benefits, the user clicks the Remove Benefit link on the Info page (Figure 2–1). The URL behind this link looks like:

  ```
  <a
  href="/empbft/controller?empID=<%=id%>&amp;action=removeBenefitFromEmployee"
  > Remove benefits from the employee</a>
  ```

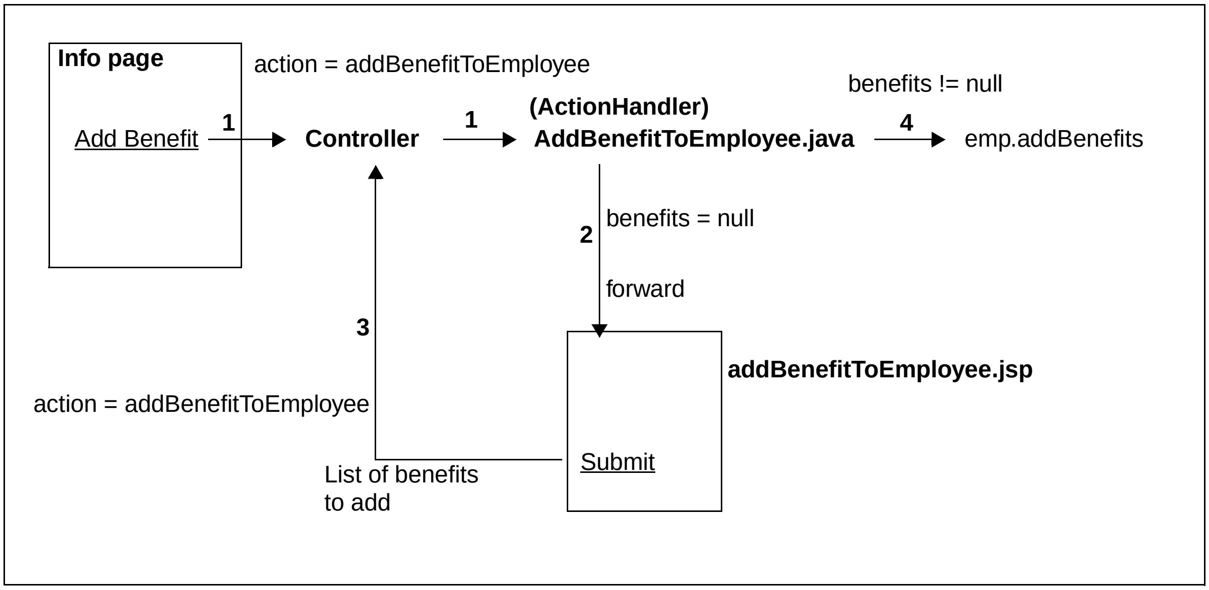
See the following sections for details on the add and remove operations.

6.4 Add Benefit Operation

6.4.1 High-Level Sequence of Events

Figure 6–4 shows the events that occur when a user selects the add benefit option:

Figure 6–4 Add Benefits Operation



1. The **Controller** servlet handles the request first. It gets the value of the **action** parameter (`addBenefitToEmployee`) and invokes the **performAction** method in the corresponding class, **AddBenefitToEmployee**.
2. The **performAction** method checks the value of the **benefits** parameter. It is null at first, so it forwards the request to **addBenefitToEmployee.jsp** (or **addBenefitToEmployeeWireless.jsp**). The JSP displays a list of benefits that the user can add. See Section 6.4.2, "Getting Benefits That the User Can Add".
3. The user selects the desired benefits to add and submits the request. The **action** parameter in the request still has the same value (`addBenefitToEmployee`), but this time, it has a **benefits** parameter that specifies which benefits to add.
4. The **Controller** invokes the **AddBenefitToEmployee** class to process the request. The class sees that the benefits parameter is not null, and it calls the

addBenefits method in the **Employee** class to add the benefits. See Section 6.4.3, "Updating the Database".

6.4.2 Getting Benefits That the User Can Add

To show a list of benefits that the user can add, the **addBenefitToEmployee.jsp** page gets a list of benefits that the user does not have. The JSP file gets an instance of **EmployeeManager**, then invokes the **getUnelectedBenefitItems** method.

```
// from addBenefitToEmployee.jsp
<%
  int empId = Integer.parseInt(request.getParameter(
                               SessionHelper.EMP_ID_PARAMETER));
  EmployeeManager mgr = SessionHelper.getEmployeeManager(request);
  Collection unelected = mgr.getUnelectedBenefitItems(empId);
  ...
%>
```

The **getUnelectedBenefitItems** method gets the master list of all benefits from **BenefitCatalog**, then it gets a list of benefits for the employee. It compares the two lists and returns a list of benefits that the employee does not have.

```
// from EmployeeManagerBean.java
public Collection getUnelectedBenefitItems(int id) throws RemoteException
{
    Collection allBenefits = null;
    InitialContext initial = new InitialContext();
    Object objref = initial.lookup(AppJNDINames.BENEFIT_CATALOG_EJBHOME);
    BenefitCatalogHome home = (BenefitCatalogHome)
          PortableRemoteObject.narrow(objref, BenefitCatalogHome.class);
    BenefitCatalog catalog = home.create();
    allBenefits = catalog.getBenefits();

    // ... exceptions omitted ...

    Collection unelected = new ArrayList();
    EmployeeModel emp = this.getEmployeeDetails(id);
    ArrayList eb = (ArrayList) emp.getBenefits();
    if (eb != null && !eb.isEmpty()) {
      Iterator i = allBenefits.iterator();
      while (i.hasNext()) {
        BenefitModel b = (BenefitModel)i.next();
        if (Collections.binarySearch(eb, b) < 0)
          unelected.add(b);
          }
          return unelected;
        }
        return allBenefits;
    }
```

6.4.3 Updating the Database

To add the benefits the user selected, the **AddBenefitToEmployee** object gets the **Employee** object and executes the **addBenefits** method:

```
// from AddBenefitToEmployee.java
String benefits[] = req.getParameterValues(SessionHelper.BENEFIT_PARAMETER);
...
int benefitIDs[] = new int[benefits.length];
for (int i = 0; i < benefits.length; i++) {
   benefitIDs[i] = Integer.parseInt(benefits[i]);
}
int empId = Integer.parseInt(req.getParameter(SessionHelper.EMP_ID_PARAMETER));
EmployeeManager mgr = SessionHelper.getEmployeeManager(req);
try {
   Employee emp = mgr.getEmployee(empId);
   emp.addBenefits(benefitIDs);
} catch (RemoteException e) {
   throw new ServletException (
          "\nRemote exception while getting employee and adding benefits.\n"
          + e.getMessage());
}
forward(req, res, wireless ? "/successWireless.jsp" : "/success.jsp");
```

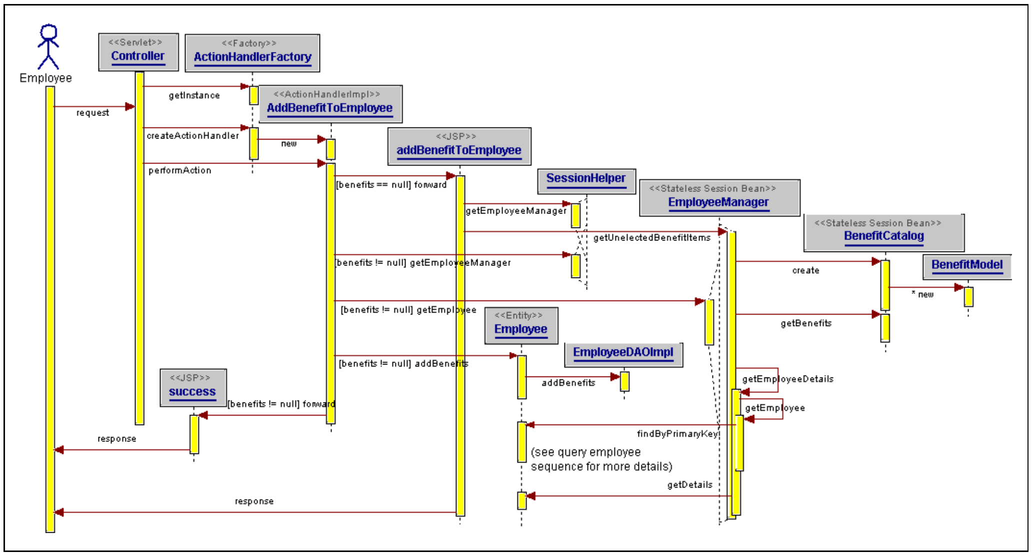
The **addBenefits** method in the **Employee** object uses the **EmployeeDAOImpl** class to connect to the database.

```
// from EmployeeBean.java
public void addBenefits(int benefits[])
{
    try{
      if (m_dao == null) m_dao = new EmployeeDAOImpl();
      m_dao.addBenefits(m_emp.getId(), benefits);
      ejbLoad();
    } catch (Exception e) {
      throw new EJBException ("\nData access exception in adding benefits.\n"
        + e.getMessage());
    }
}
```

After adding the benefits in the database, the **addBenefits** method calls the **ejbLoad** method to synchronize the **Employee** bean with the data in the database.

The **addBenefits** method in **EmployeeDAOImpl** connects to the database and sends an INSERT statement.

Figure 6–5 Sequence Diagram for Adding Benefits

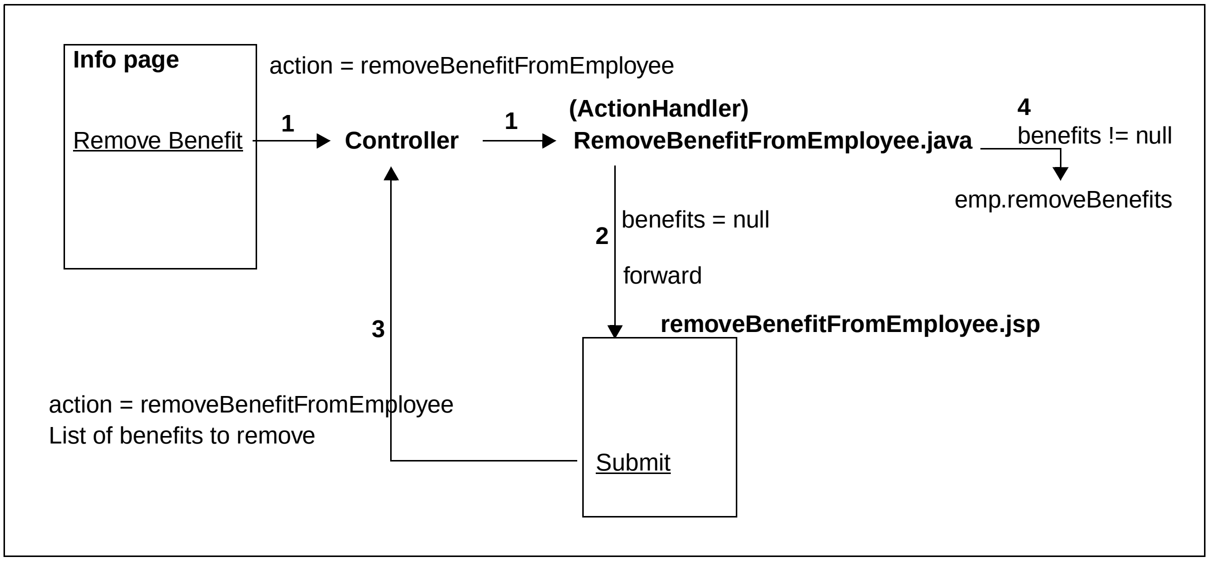

Employee
<<Servlet>> Controller
<<Factory>> ActionHandlerFactory
<<ActionHandlerImpl>> AddBenefitToEmployee
<<JSP>> addBenefitToEmployee
SessionHelper
<<Stateless Session Bean>> EmployeeManager
<<Stateless Session Bean>> BenefitCatalog
BenefitModel
<<Entity>> Employee
EmployeeDAOImpl
<<JSP>> success
request
getInstance
createActionHandler
new
performAction
[benefits == null] forward
getEmployeeManager
getUnelectedBenefitItems
create
* new
[benefits != null] getEmployeeManager
[benefits != null] getEmployee
getBenefits
[benefits != null] addBenefits
addBenefits
getEmployeeDetails
getEmployee
[benefits != null] forward
findByPrimaryKey
(see query employee sequence for more details)
getDetails
response
response

6.5 Removing Benefit Operation

6.5.1 High-Level Sequence of Events

Figure 6–6 shows the events that occur when a user selects some benefits to remove and clicks the Submit button.

Figure 6–6 Remove Benefits Operation


Info page
Remove Benefit
action = removeBenefitFromEmployee
1
Controller
1
(ActionHandler)
RemoveBenefitFromEmployee.java
4
benefits != null
emp.removeBenefits
2
benefits = null
forward
removeBenefitFromEmployee.jsp
3
action = removeBenefitFromEmployee
List of benefits to remove
Submit

1. The **Controller** servlet handles the request first. It gets the value of the **action** parameter (`removeBenefitFromEmployee`) and invokes the **performAction** method in the corresponding class, **RemoveBenefitFromEmployee**.
2. The **performAction** method checks the value of the **benefits** parameter. It is null at first, so it forwards the request to **removeBenefitFromEmployee.jsp** (or **removeBenefitFromEmployeeWireless.jsp**). The JSP displays a list of benefits that the user can remove. See Section 6.5.2, "Getting Benefits That the User Can Remove".
3. The user selects the desired benefits to remove and submits the request. The **action** parameter in the request still has the same value (`removeBenefitFromEmployee`), but this time, it has a **benefits** parameter that specifies which benefits to remove.

4. The **Controller** invokes the **RemoveBenefitFromEmployee** class to process the request. The class sees that the benefits parameter is not null, and it calls the **removeBenefits** method in the **Employee** class to remove the benefits. See Section 6.4.3, "Updating the Database".

6.5.2 Getting Benefits That the User Can Remove

To get a list of benefits that the user can remove, the **removeBenefitFromEmployee.jsp** gets an **EmployeeModel**, which contains all the data for an employee, then it calls the **getBenefits** method in **EmployeeModel**. It then iterates through the list to display each benefit.

```
<%
  int empId = Integer.parseInt(request.getParameter(
                              SessionHelper.EMP_ID_PARAMETER));
  EmployeeManager mgr = SessionHelper.getEmployeeManager(request);
  Collection elected = mgr.getEmployeeDetails(empId).getBenefits();
  if (elected == null || elected.size() == 0) {
%>
<h4>No Benefits to Remove</h4>
<p>The employee has not elected any benefits.</p>
<h4>Actions</h4>
<a href="./controller?action=queryEmployee&amp;empID=<%=empId%>">Query the same
employee</a><br>
<a href="./controller?action=queryEmployee">Query other employee</a><br>
<a href="./">Home</a><br>
<%
  } else {
%>
<h4>Select Elected Benefits</h4>
<%
    Iterator i = elected.iterator();
    while (i.hasNext()) {
        BenefitItem b = (BenefitItem) i.next();
%>
<input type=checkbox name=benefits value=<%=b.getId()%>><%=b.getName()%><br>
<%
    } // End of while
%>
<h4>Actions</h4>
<input type=submit value="Remove Selected Benefits">
<input type=hidden name=empID value=<%=empId%>>
<input type=hidden name=action
          value=<%=SessionHelper.ACTION_REMOVE_BENEFIT_FROM_EMPLOYEE%>>
<%
  } // End of if
%>
```

6.5.3 Updating the Database

To remove the benefits the user selected, the **RemoveBenefitFromEmployee** object gets the **Employee** object and executes the **removeBenefits** method:

```
// from RemoveBenefitFromEmployee.java
String benefits[] = req.getParameterValues(SessionHelper.BENEFIT_PARAMETER);
String client = req.getParameter(SessionHelper.CLIENT_TYPE_PARAMETER);
boolean wireless = client != null &&
                client.equals(SessionHelper.CLIENT_TYPE_WIRELESS);
if(benefits == null) {
   forward(req, res, wireless ?
           "/removeBenefitFromEmployeeWireless.jsp" :
           "/removeBenefitFromEmployee.jsp");
} else {
   int benefitIDs[] = new int[benefits.length];
   for (int i = 0; i < benefits.length; i++) {
      benefitIDs[i] = Integer.parseInt(benefits[i]);
   }
   int empId = Integer.parseInt(req.getParameter(
                              SessionHelper.EMP_ID_PARAMETER));
   EmployeeManager mgr = SessionHelper.getEmployeeManager(req);
   try {
      Employee emp = mgr.getEmployee(empId);
      emp.removeBenefits(benefitIDs);
   } catch (RemoteException e) {
      throw new ServletException (
        "Remote exception while getting employee and removing his/her
            benefits." + e.getMessage());
   }
   forward(req, res, wireless ? "/successWireless.jsp" : "/success.jsp");
}
```

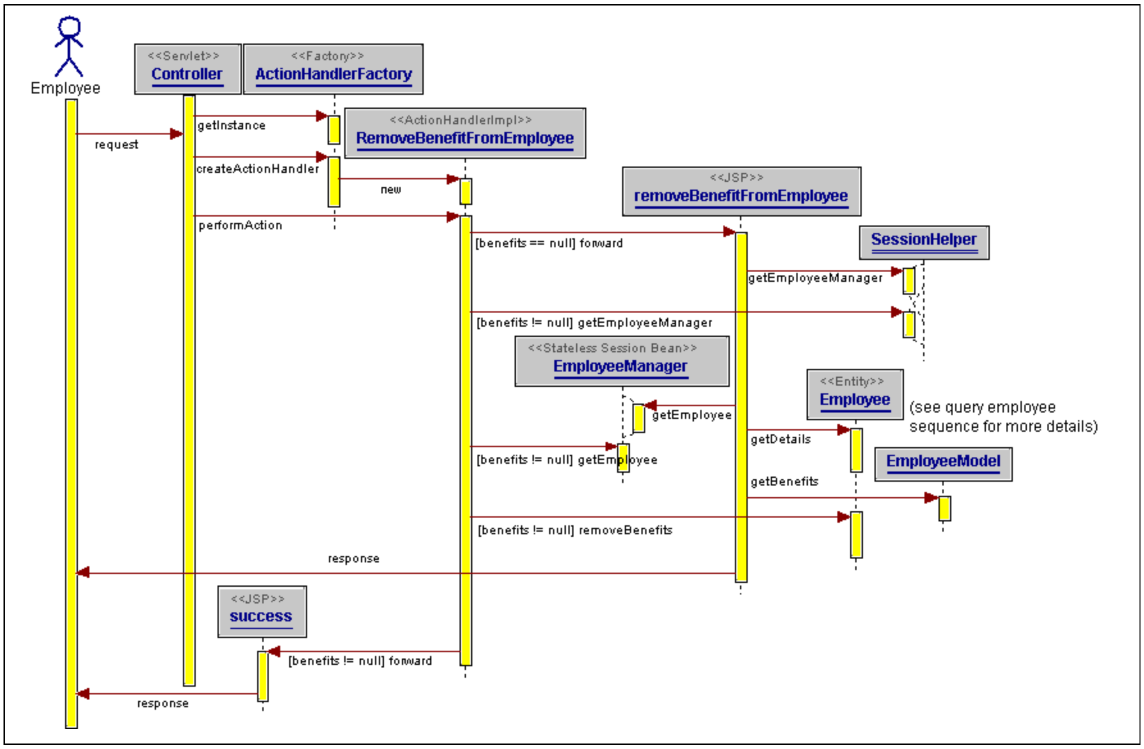
The **removeBenefits** method in the **Employee** object uses the **EmployeeDAOImpl** class to connect to the database.

```
// from EmployeeBean.java
public void removeBenefits(int benefits[])
{
      try {
         if (m_dao == null)  m_dao = new EmployeeDAOImpl();
         m_dao.removeBenefits(m_emp.getId(), benefits);
         ejbLoad();
       } catch (Exception e) {
         throw new EJBException ("\nData access exception in removing benefits.\n"
           + e.getMessage());
      }
   }
```

After removing the benefits from the database, the **removeBenefits** method calls the **ejbLoad** method to synchronize the **Employee** bean with the data in the database.

The **removeBenefits** method in **EmployeeDAOImpl** connects to the database and sends a DELETE statement.

Figure 6–7 Sequence Diagram for Removing Benefits



7 Supporting Wireless Clients

The wireless feature in Oracle9*i*AS enables wireless clients to access your applications. Because wireless clients use protocols different from HTTP and markup languages other than HTML, you have to make some modifications to your application to support wireless clients.

Contents of this chapter:

- Section 7.1, "Changes You Need To Make To Your Application"
- Section 7.2, "Presentation Data for Wireless Clients"
- Section 7.3, "Deciding Where to Put the Presentation Data for Wireless Clients"
- Section 7.4, "Header Information in JSP Files for Wireless Clients"
- Section 7.5, "Operation Details"
- Section 7.6, "Accessing the Application"

7.1 Changes You Need To Make To Your Application

If your application uses the MVC design, you only need to make a few changes to your application to support wireless clients:

- The major change you have to make to your application to support wireless clients is to write the presentation data for the wireless clients. The business logic objects remain unchanged.

  This task is simplified by the separation of the presentation data from the business logic objects. If there were no clear separation between presentation data and business logic objects, you would have more difficulty merging presentation code for wireless clients with presentation code for desktop browsers.

  See Section 7.2, "Presentation Data for Wireless Clients".

- You may also have to modify the objects that subclass the **ActionHandler** object (see Figure 4–1). These objects forward the request to the appropriate JSP files. When you write your presentation data for wireless clients, you may choose to put the data in the same JSP file that contains the presentation data for browsers, or in different JSP files. If you choose to put the data in separate files, then you have to edit the **ActionHandler** objects to forward requests from wireless clients to JSP files that contain wireless presentation data.

  See Section 7.3.3, "Separating Presentation Data into Separate Files".

7.2 Presentation Data for Wireless Clients

Because wireless clients do not use a standardized markup language, you have to write presentation data for the clients in XML based on a generic DTD specification. The wireless feature in Oracle9*i*AS transforms the XML to the specific markup language that the wireless client can process.

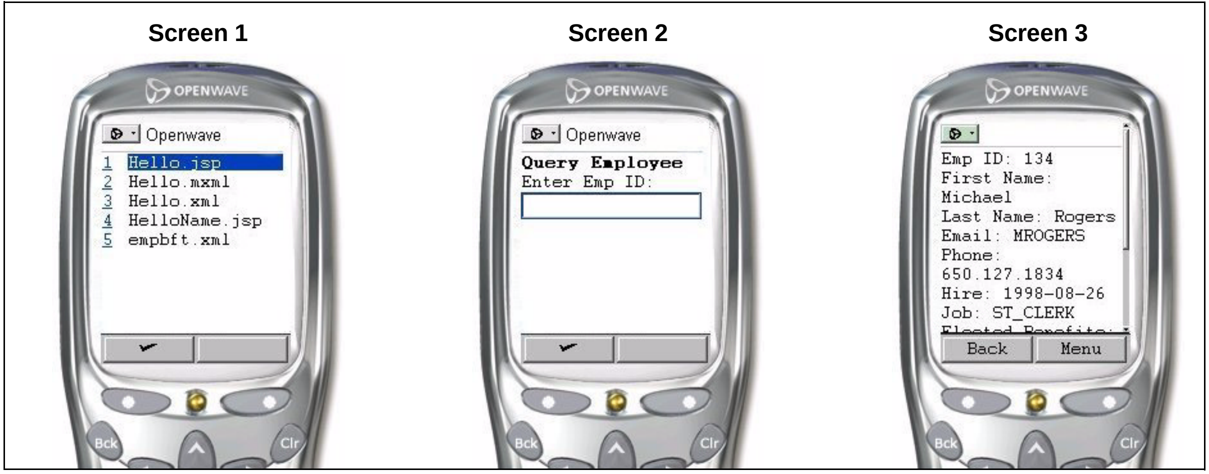
Like HTML, applications can generate XML from JSP files or static files. In the sample application, the presentation data comes from JSP files because it contains dynamic data. See Chapter 5, "Creating Presentation Pages".

The generic XML for wireless clients is based on the SimpleResult DTD. For details on the DTD and how to use its elements, see the *Oracle9i*AS *Wireless Developer's Guide*.

7.2.1 Screens for the Wireless Application

Figure 7–1 to Figure 7–3 show the Employee Benefit sample application on an OpenWave simulator. The application on a wireless client looks similar to the application on a desktop browser.

Figure 7–1 Screens for the wireless application (1 of 3)



```
1 Hello.jsp
2 Hello.mxml
3 Hello.xml
4 HelloName.jsp
5 empbft.xml
```

```
Query Employee
Enter Emp ID:
```

```
Emp ID: 134
First Name:
Michael
Last Name: Rogers
Email: MROGERS
Phone:
650.127.1834
Hire: 1998-08-26
Job: ST_CLERK
```

On Screen 1, the wireless client lists the applications that it can run. This is essentially a list of the files in:

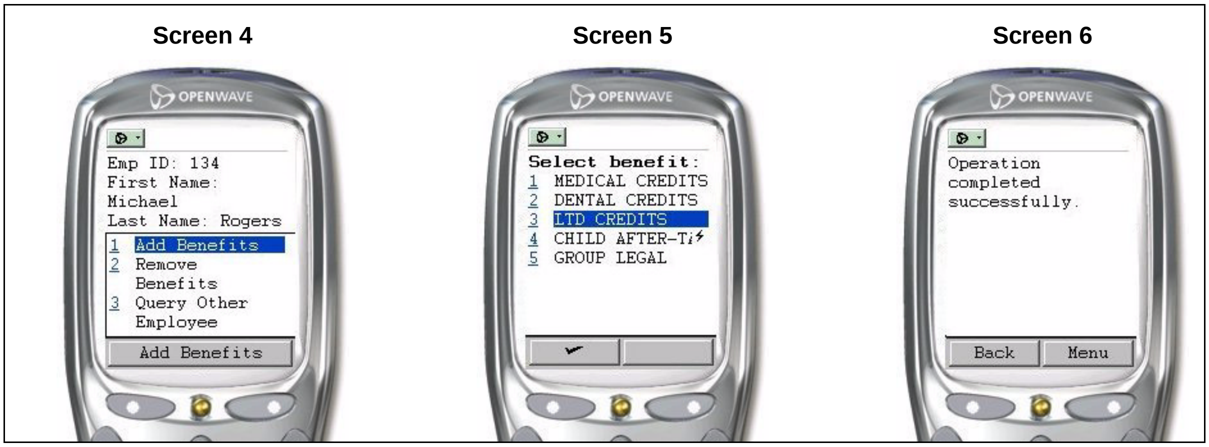
```
$OMSDK_HOME/oc4j_omsdk/omsdk/j2ee/applications/pmsdk/apps/
```

$OMSDK_HOME is the home directory for Oracle9*i*AS Wireless SDK.

Screen 2 shows the sample application's starting point, which is the **empbft.xml** file. The file displays a text input field where the user can enter an employee ID.

Screen 3 shows the results of the query. The wireless client has a scrollbar that enables the user to scroll down the page to view all the information. At this screen, the user can press the Menu button to add or remove benefits.

Figure 7–2 Screens for the wireless application (2 of 3)



```
Emp ID: 134
First Name:
Michael
Last Name: Rogers
1 Add Benefits
2 Remove
  Benefits
3 Query Other
  Employee
Add Benefits
```

```
Select benefit:
1 MEDICAL CREDITS
2 DENTAL CREDITS
3 LTD CREDITS
4 CHILD AFTER-Ti
5 GROUP LEGAL
```

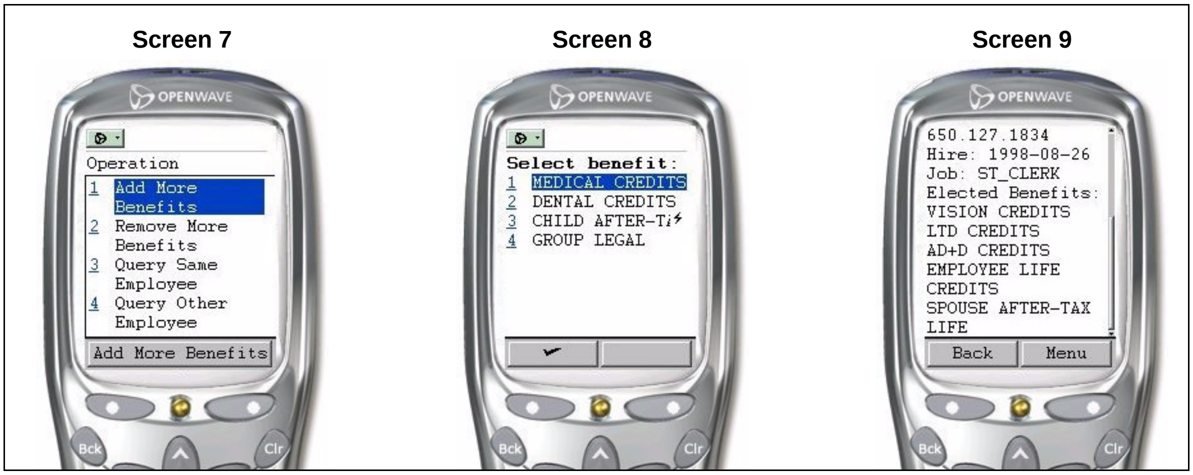
```
Operation
completed
successfully.
Back    Menu
```

Screen 4 shows the menu, which offers selections such as add benefits, remove benefits, and query other employee.

Screen 5 shows a list of benefits that the user can add. The user selects one benefit to add and clicks OK to submit the request. Note that on wireless clients, the user can select only one item to add or remove at a time. See Section 7.2.2, "Differences Between the Wireless and the Browser Application".

Screen 6 tells the user that the add benefit operation was completed successfully. This screen also has a Menu option.

Figure 7–3 Screens for the wireless application (3 of 3)



650.127.1834
Hire: 1998-08-26
Job: ST_CLERK
Elected Benefits:
VISION CREDITS
LTD CREDITS
AD+D CREDITS
EMPLOYEE LIFE CREDITS
SPOUSE AFTER-TAX LIFE

Screen 7 shows the menu. It has four options: add more benefits, remove more benefits, query same employee, and query other employee.

Screen 8 shows the list of benefits after the user has added a benefit.

Screen 9 is similar to Screen 3, except that it is scrolled down to show the list of benefits for the user.

7.2.2 Differences Between the Wireless and the Browser Application

In the browser version of the Employee Benefit application, users can select multiple benefits to add or remove. On wireless devices, however, users can select only one item at a time. To assist users in adding/removing multiple items, the application provides options called "Add More Benefits" and "Remove More Benefits" to enable users to select another benefit to add or remove (screen 7). These options are not available on browsers.

These options are made available from **successWireless.jsp**, which is displayed after the application adds or removes a benefit successfully (screen 6). This screen displays a success message. When users click Menu on this screen, they see the "Add More Benefits" and "Remove More Benefits" options.

```
// from successWireless.jsp
<SimpleText>
   <SimpleTextItem>Operation completed successfully.</SimpleTextItem>
   <Action label="Add More Benefits" type="SOFT1" task="GO"
target="/empbft/controller?action=addBenefitToEmployee&amp;clientType=wireless&a
```

```
mp;empID=<%=empId%>"></Action>

   <Action label="Remove More Benefits" type="SOFT1" task="GO"
target="/empbft/controller?action=removeBenefitFromEmployee&amp;clientType=wirel
ess&amp;empID=<%=empId%>"></Action>

   <Action label="Query Same Employee" type="SOFT1" task="GO"
target="/empbft/controller?action=queryEmployee&amp;clientType=wireless&amp;empI
D=<%=empId%>"></Action>

   <Action label="Query Other Employee" type="SOFT1" task="GO"
target="/empbft/controller?action=queryEmployee&amp;clientType=wireless"></Actio
n>
</SimpleText>
```

When the user selects the "Add More Benefits" or "Remove More Benefits" option, the request is similar to the request to add or remove a benefit. The request contains an `action` parameter, an `empID` parameter, and a `clientType` parameter. The application requeries the database and displays an updated list of benefits (Screen 8).

7.3 Deciding Where to Put the Presentation Data for Wireless Clients

You can write the XML presentation data for wireless clients in the same JSP file as the one that generates the HTML, or in a different JSP file. Regardless of where you put the presentation data, you still need to determine if a request came from a wireless or desktop client.

7.3.1 Determining the Origin of a Request

You can determine the origin of a request by inserting a parameter in the request to identify wireless clients. You can include the parameter and its value using a hidden input form element.

The sample application uses a parameter name of `clientType` and parameter value of `wireless` to identify wireless clients. Each wireless client request contains this parameter. For example, in **empbft.xml**, which is the first file in the application that wireless clients see:

```
// empbft.xml
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<SimpleResult>
  <SimpleContainer>
    <SimpleForm title="Query Employee" target="/empbft/controller">
       <SimpleFormItem name="empID" format="*N">Enter Emp ID: </SimpleFormItem>
       <SimpleFormItem name="action" type="hidden" value="queryEmployee" />
       <SimpleFormItem name="clientType" type="hidden" value="wireless" />
    </SimpleForm>
  </SimpleContainer>
</SimpleResult>
```

You can then check for the `clientType` parameter in servlets or JSPs in the same way that you check for other parameters.

In servlets:

```
String client = req.getParameter(SessionHelper.CLIENT_TYPE_PARAMETER);
boolean wireless =
          client != null && client.equals(SessionHelper.CLIENT_TYPE_WIRELESS);
```

In JSPs:

```
<%
  String client = request.getParameter(SessionHelper.CLIENT_TYPE_PARAMETER);
  boolean wireless =
            client != null && client.equals(SessionHelper.CLIENT_TYPE_WIRELESS);
%>
```

7.3.2 Combining Presentation Data in the Same JSP File

If you use this method, determine the origin of the request (whether it came from a wireless or desktop client) in the JSP file itself. You can then generate HTML or XML depending on the origin. For example:

```
// import classes for both wireless and browsers
<%@ page import="java.util.*" %>
<%@ page import="empbft.component.employee.ejb.*" %>
<%@ page import="empbft.component.employee.helper.*" %>
<%@ page import="empbft.util.*" %>

// check the client type that sent the request
<%
  String client = request.getParameter(SessionHelper.CLIENT_TYPE_PARAMETER);
  boolean wireless = ( (client != null) &&
                client.equals(SessionHelper.CLIENT_TYPE_WIRELESS) );
  if (wireless)
  {
%>
    <?xml version = "1.0" encoding = "ISO-8859-1"?>
    <%@ page contentType="text/vnd.oracle.mobilexml; charset=ISO-8859-1" %>
    <SimpleResult>
      <SimpleContainer>
        <SimpleForm title="Query Employee" target="/empbft/controller">
           <SimpleFormItem name="empID" format="*N">Enter Emp ID:
                                                    </SimpleFormItem>
           <SimpleFormItem name="action" type="hidden" value="queryEmployee" />
           <SimpleFormItem name="clientType" type="hidden" value="wireless" />
        </SimpleForm>
      </SimpleContainer>
    </SimpleResult>
<%
  } else {
%>
    <%@ page contentType="text/html;charset=ISO-8859-1"%>
    <html>
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <link rel="stylesheet" href="css/blaf.css" type="text/css">
    <title>Query Employee</title>
    </head>
    <body>
    <h2>Employee Benefit Application</h2>
<%
  String empId = request.getParameter(SessionHelper.EMP_ID_PARAMETER);
  if (empId == null)
  {
%>
<h4>Query Employee</h4>
<form method=get action="/empbft/controller">
<input type=hidden name=action value=queryEmployee>
<table>
  <tr>
    <td>Employee ID:</td>
    <td><input type=text name=empID size=4></td>
    <td><input type=submit value="Query Employee"></td>
  </tr>
</table>
<h4>Actions</h4>
<a href="/empbft/">Home</a><br>
</form>
<%
  } else {
    int id = Integer.parseInt(empId);
    EmployeeManager mgr = SessionHelper.getEmployeeManager(request);
    EmployeeModel emp = mgr.getEmployeeDetails(id);
%>
<h4>Employee Details</h4>
<table>
<tr><td>Employee ID: </td><td colspan=3><b><%=id%></b></td></tr>
<tr><td>First Name: </td><td><b><%=emp.getFirstName()%></b></td><td>Last Name:
</td><td><b><%=emp.getLastName()%></b></td></tr>
<tr><td>Email: </td><td><b><%=emp.getEmail()%></b></td><td>Phone Number:
</td><td><b><%=emp.getPhoneNumber()%></b></td></tr>
<tr><td>Hire Date:
</td><td><b><%=emp.getHireDate().toString()%></b></td><td>Job:
</td><td><b><%=emp.getJobId()%></b></td></tr>
</table>
<h4>Elected Benefits</h4>
<table>
  <%
    Collection benefits = emp.getBenefits();
    if (benefits == null || benefits.size() == 0) {
  %>
      <tr><td>None</td></tr>
  <%
    } else {
      Iterator it = benefits.iterator();
      while (it.hasNext()) {
        BenefitItem item = (BenefitItem)it.next();
  %>
        <tr><td><%=item.getName()%></td></tr>
  <%
      } // end of while
    } // end of if
  %>
</table>
<h4>Actions</h4>
<table>
<tr><td><a
href="/empbft/controller?empID=<%=id%>&amp;action=addBenefitToEmployee">Add
benefits to the employee</a></td></tr>
<tr><td><a
href="/empbft/controller?empID=<%=id%>&amp;action=removeBenefitFromEmployee">Rem
ove benefits from the employee</a></td></tr>
<tr><td><a href="/empbft/controller?action=queryEmployee">Query other
employee</a></td></tr>
<tr><td><a href="/empbft/">Home</a><br></td></tr>
</table>
<%
    } // end of else (empId != null)
%>
</body>
</html>
<%
} // end of else (wireless)
%>
```

7.3.3 Separating Presentation Data into Separate Files

If you are using different files, edit the subclasses of **ActionHandler** to check the origin of the request, and forward the request to the proper JSP file. For example:

```
public void performAction(HttpServletRequest req, HttpServletResponse res)
    throws ServletException
{
   String client = req.getParameter(SessionHelper.CLIENT_TYPE_PARAMETER);
   boolean wireless =
           client != null && client.equals(SessionHelper.CLIENT_TYPE_WIRELESS);
   String empIdString = req.getParameter(SessionHelper.EMP_ID_PARAMETER);
   boolean validEmpId = true;
   if (empIdString != null) {
      int empId = Integer.parseInt(empIdString);
      validEmpId = (empId >= 100 && empId <= 206) ? true : false;
   }

   // Forward to appropriate page
   if (wireless) {
      if (validEmpId) {
         forward(req, res, "/queryEmployeeWireless.jsp");
      } else {
         forward(req, res, "/errorWireless.jsp");
      }
   } else {
      if (validEmpId) {
         forward(req, res, "/queryEmployee.jsp");
      } else {
         forward(req, res, "/error.jsp");
      }
   }
}
```

The value of CLIENT_TYPE_PARAMETER is defined in **SessionHelper** to be clientType. This is the name of the parameter.

The value of CLIENT_TYPE_WIRELESS is defined in **SessionHelper** to be wireless. This is the value of the parameter.

This parameter and the value of the parameter are defined in **empbft.xml**. This file corresponds to the ID page for wireless. It enables users to enter a number in a text input field.

```
// empbft.xml
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<SimpleResult>
  <SimpleContainer>
    <SimpleForm title="Query Employee" target="/empbft/controller">
       <SimpleFormItem name="empID" format="*N">Enter Emp ID: </SimpleFormItem>
       <SimpleFormItem name="action" type="hidden" value="queryEmployee" />
       <SimpleFormItem name="clientType" type="hidden" value="wireless" />
    </SimpleForm>
  </SimpleContainer>
</SimpleResult>
```

7.4 Header Information in JSP Files for Wireless Clients

You have to make some changes in the header of your JSP files for wireless clients:

7.4.1 Setting the XML Type

The first line of the JSP file should specify that the file is an XML file:

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
```

7.4.2 Setting the Content Type

In the JSP files for wireless clients, you need the following line at the top of the files to set the content type of the response and the character set.

```
<%@ page contentType="text/vnd.oracle.mobilexml; charset=ISO-8859-1" %>
```

You need to do this because the default value for `contentType` for JSPs is `text/html`, and this is not what you want for wireless clients.

The transformer uses the `text/vnd.oracle.mobilexml` value when transforming the page into data that the wireless client can understand.

7.5 Operation Details

To Oracle9*i*AS, requests from wireless clients look the same as requests from desktop browsers except that the user agent field contains the name of the wireless device. However, the way in which wireless requests get to Oracle9*i*AS is different: Wireless requests first go through gateways (such as WAP, Voice, or SMS), which convert the requests to the HTTP protocol. The gateways then route the requests to Oracle9*i*AS Wireless.

Oracle9*i*AS Wireless processes the requests by invoking an adapter to retrieve XML from the mobile application. The XML is based on a schema defined by Oracle9*i*AS.

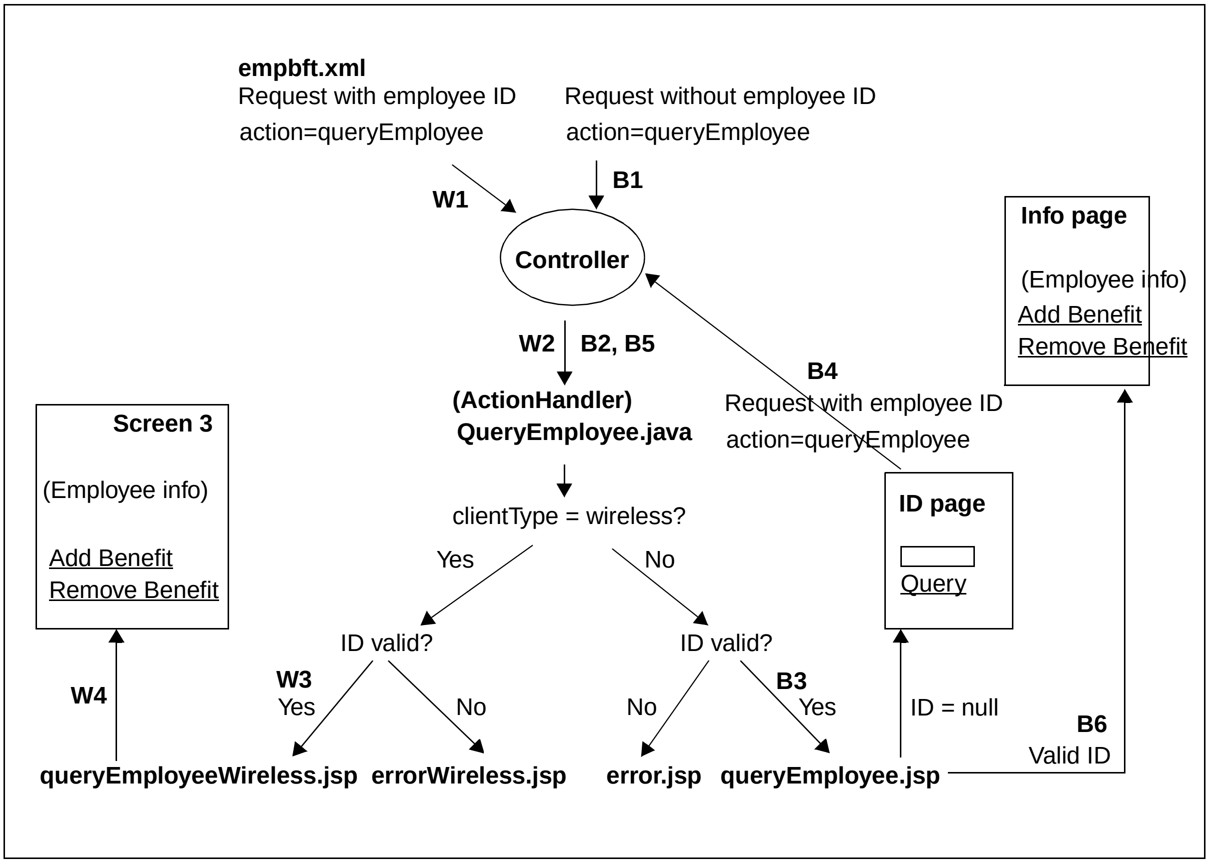
Oracle9*i*AS Wireless then invokes a transformer, which takes the XML and transforms it to a markup language appropriate for the wireless client. Oracle9*i*AS sends the resulting data to the gateway, which may encode the data (to make the data more compact) before sending it to the client.

See the *Oracle9i*AS *Wireless Developer's Guide* and the *Oracle9i*AS *Getting Started and System Guide* for further details.

7.5.1 Query Operation

Figure 7–4 shows the flow of the query operation with wireless and browser clients. This figure is a more complex form of Figure 6–1.

Figure 7–4 Query Operation



The figure above contains two sequences of events. One sequence is for requests that come from browsers; steps in this sequence are noted in the figure with a "B". The other sequence is for requests that come from wireless clients; steps in this sequence are noted with a "W".

The steps for browser requests are covered in Section 6.2, "Query Employee Operation". This section covers the wireless steps.

W1: The server sends the request to the Controller with the `action` parameter set to `queryEmployee` and the `empID` parameter set to the employee ID entered by the user.

W2: **QueryEmployee.java** checks the `clientType` parameter to determine if the request came from a wireless client or a browser. This parameter is set only in the

XML files that the application sends to wireless clients; requests from browsers do not have this parameter. **QueryEmployee.java** also checks if the employee ID is valid.

W3: **QueryEmployee.java** forwards the request to **queryEmployeeWireless.jsp**.

W4: **queryEmployeeWireless.jsp** is similar to **queryEmployee.jsp**. It retrieves and displays employee data. Note that the retrieval method is the same in both files. The only difference is in the tags used (HTML for browsers, XML for wireless clients).

7.5.2 queryEmployeeWireless.jsp

queryEmployeeWireless.jsp looks like the following:

```
// queryEmployeeWireless.jsp
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<%@ page contentType="text/vnd.oracle.mobilexml; charset=ISO-8859-1" %>
<%@ page import="java.util.*" %>
<%@ page import="empbft.component.employee.ejb.*" %>
<%@ page import="empbft.component.employee.helper.*" %>
<%@ page import="empbft.util.*" %>
<SimpleResult>
   <SimpleContainer>
<%
  String empId = request.getParameter(SessionHelper.EMP_ID_PARAMETER);
  if (empId == null)
  {
%>
     <SimpleForm title="Query Employee" target="/empbft/controller">
       <SimpleFormItem name="empID" format="*N">Enter Emp ID: </SimpleFormItem>
       <SimpleFormItem name="action" type="hidden" value="queryEmployee" />
       <SimpleFormItem name="clientType" type="hidden" value="wireless" />
     </SimpleForm>
<%
  } else {
    int id = Integer.parseInt(empId);
    EmployeeManager mgr = SessionHelper.getEmployeeManager(request);
    EmployeeModel emp = mgr.getEmployeeDetails(id);
%>
     <SimpleText>
       <SimpleTextItem>Emp ID: <%=empId%></SimpleTextItem>
       <SimpleTextItem>First Name: <%=emp.getFirstName()%></SimpleTextItem>
       <SimpleTextItem>Last Name: <%=emp.getLastName()%></SimpleTextItem>
       <SimpleTextItem>Email: <%=emp.getEmail()%></SimpleTextItem>
       <SimpleTextItem>Phone: <%=emp.getPhoneNumber()%></SimpleTextItem>
       <SimpleTextItem>Hire: <%=emp.getHireDate()%></SimpleTextItem>
       <SimpleTextItem>Job: <%=emp.getJobId()%></SimpleTextItem>
       <SimpleTextItem>Elected Benefits: </SimpleTextItem>
<%
    Collection benefits = emp.getBenefits();
    if (benefits == null || benefits.size() == 0) {
%>
       <SimpleTextItem>None</SimpleTextItem>
<%
    } else {
      Iterator it = benefits.iterator();
      while (it.hasNext()) {
        BenefitItem item = (BenefitItem)it.next();
%>
       <SimpleTextItem><%=item.getName()%></SimpleTextItem>
<%
      } // end of while
    } // end of if
%>
     <Action label="Add Benefits" type="SOFT1" task="GO"
           target="/empbft/controller?action=addBenefitToEmployee&amp;
                    clientType=wireless&amp;empID=<%=empId%>"></Action>
     <Action label="Remove Benefits" type="SOFT1" task="GO"
             target="/empbft/controller?action=removeBenefitFromEmployee&amp;
             clientType=wireless&amp;empID=<%=empId%>"></Action>
     <Action label="Query Other Employee" type="SOFT1" task="GO"
                  target="/empbft/controller?action=queryEmployee&amp;
                  clientType=wireless"></Action>
     </SimpleText>
<%
    } // end of else (empId != null)
%>
   </SimpleContainer>
</SimpleResult>
```

The `Action` tag defines popup menus (Figure 7–1, Screen 4). The user presses the Menu button to access the popup menu.

7.5.3 Add and Remove Benefits Operations

The add and remove benefits operations for wireless clients are similar to the corresponding operations for browsers. The changes in the application needed to support these operations for wireless clients include:

- Modifying **AddBenefitToEmployee.java** and **RemoveBenefitFromEmployee.java** to check if the request came from a wireless client. The checks use the same format as in the query operation.
- Creating **addBenefitToEmployeeWireless.jsp** and **removeBenefitFromEmployeeWireless.jsp** to define the XML for presentation.
- Creating **errorWireless.jsp** to display an error message.
- Creating **successWireless.jsp**, which the application displays when a user successfully adds or removes a benefit. In addition to displaying a success message, the file also defines a popup menu that enables the user to add or remove additional benefits without having to go to the main menu. This feature is not applicable to browsers. Section 7.2.2, "Differences Between the Wireless and the Browser Application" describes this feature in detail.

7.6 Accessing the Application

While you are developing wireless applications, you may not have access to an environment where you can run your applications from actual wireless clients. In such cases, you can test your applications using simulators. However, before you deploy your applications in a production environment, it is highly recommended that you find or set up an environment where you can test your applications with actual wireless clients.

7.6.1 Using a Simulator

To access the application from a wireless client simulator:

1. Enter the following URL in the simulator:

   ```
   http://<host>:<port>/omsdk/rm
   ```

   /omsdk/rm points to the wireless application. It displays a screen with two choices:

   - Go To ...

     This selection displays a screen with a text field that enables you to enter a URL to visit.

   - Samples

     This selection displays a screen (Figure 7–1, Screen 1) that lists all the applications in a certain directory (see Section 7.2.1, "Screens for the Wireless Application").

2. Select Samples.
3. Invoke your application from the list of applications.

7.6.2 Using an Actual Wireless Client

To access the application from an actual web-enabled wireless client such as a cell phone or PDA, check that you have the following:

- Oracle9*i*AS that hosts the application is running on a machine that is outside of any firewalls.
- The database that the application requires is also running outside of any firewalls.

Oracle9*i*AS and the database need to be freely accessible because requests from wireless clients go through gateways, which can communicate only with machines that are outside of any firewalls.

Your application should then appear on the list of applications when you enter the URL and follow the steps listed in Section 7.6.1, "Using a Simulator".

8 Adding Web Cache to the Application

You can use the web cache feature of Oracle9*i*AS to improve performance, availability, and scalability of your applications without modifying them. You just have to specify which pages in your applications you want to cache using the Oracle Web Cache Manager tool.

- Section 8.1, "Choosing Which Pages to Cache"
- Section 8.2, "Analyzing the Application"

This guide does not cover how web cache works. For an overview and details of web cache, see the *Oracle9i*AS *Web Cache Administration and Deployment Guide*.

8.1 Choosing Which Pages to Cache

Pages that you should cache include the following:

- Static pages such as HTML, XML, and text pages
- Style sheets such as CSS style sheets
- Graphics, which are generally static
- PDF files
- Dynamic pages

Note: If you cache dynamic pages, be careful to invalidate them when the data in the data source changes. Otherwise, users may get outdated pages from the cache.

You use the Oracle Web Cache Manager to manage cached pages. This applies to static and dynamic elements. To cache a page, you specify the page’s URL in the Oracle Web Cache Manager. You can use regular expressions to match multiple URLs and to ensure your pattern matches exactly.

The next section shows how the Employee Benefit application caches a combination of static pages and dynamic pages.

8.2 Analyzing the Application

The only static element in the Employee Benefit application is a style sheet (blaf.css).

The ID page (Figure 2–1), which prompts the user to enter an employee ID, is a static page in the sense that it does not change from user to user, but it is generated dynamically. This is a good page to cache.

The most requested pages in the application are the pages that display employee information. Caching these pages would improve application performance. These pages are dynamically generated, however; the application needs to invalidate them when they are no longer valid.

There are no graphics to cache in this application.

8.2.1 Specifying the Pages to Cache

Figure 8–1 shows the Oracle Web Cache Manager with the Cacheability Rules page selected. The first three lines are specific to the Employee Benefit application.

The ID page and the pages that display employee information have similar URLs. The URL for the ID page is:

```
/empbft/controller?action=queryEmployee
```

The employee information pages have URLs that look something like this:

```
/empbft/controller;jsessionid=489uhhjjhjkui348fslkj0982k3jlds3?action=queryEmplo
yee&empID=123&submit=Query+Employee
```

Both URLs have `action=queryEmployee`. The following regular expression covers both URLs:

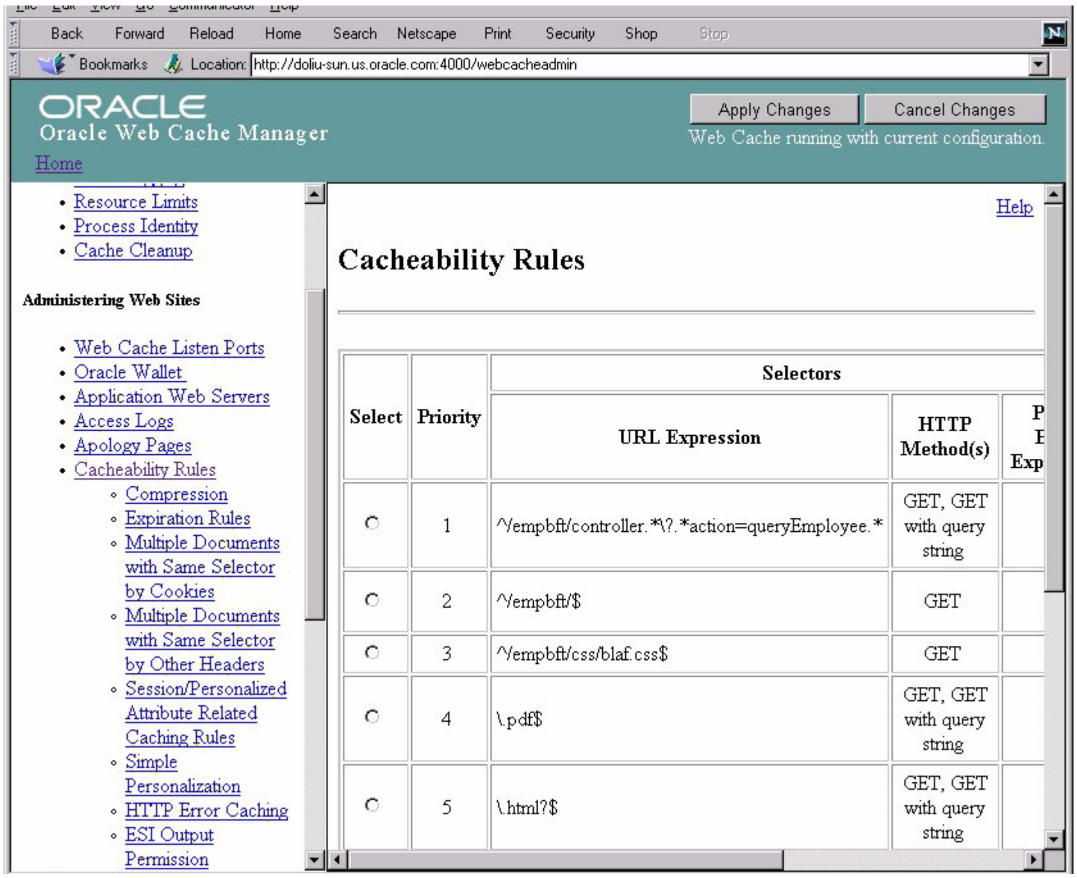
```
^/empbft/controller.*\?.*action=queryEmployee.*
```

To cache the style sheet, specify its URL in the Oracle Web Cache Manager. The ^ and $ are special characters used in regular expressions to indicate the beginning and the end of a line. This ensures that the pattern matches exactly.

```
^/empbft/css/blaf.css$
```

The second rule, which matches `^/empbft/$`, specifies an optional convenience page that provides a link to the ID page of the application. This page is static. If you have a page external to the application that links to the application, then you do not need this page and URL.

Figure 8–1 Oracle Web Cache Manager



Web Cache running with current configuration.

- Resource Limits
- Process Identity
- Cache Cleanup

Administering Web Sites

- Web Cache Listen Ports
- Oracle Wallet
- Application Web Servers
- Access Logs
- Apology Pages
- Cacheability Rules
  - Compression
  - Expiration Rules
  - Multiple Documents with Same Selector by Cookies
  - Multiple Documents with Same Selector by Other Headers
  - Session/Personalized Attribute Related Caching Rules
  - Simple Personalization
  - HTTP Error Caching
  - ESI Output Permission

Cacheability Rules

Select	Priority	Selectors		
		URL Expression	HTTP Method(s)	P E Exp
○	1	^/empbft/controller.*\?.*action=queryEmployee.*	GET, GET with query string	
○	2	^/empbft/$	GET	
○	3	^/empbft/css/blaf.css$	GET	
○	4	\.pdf$	GET, GET with query string	
○	5	\.html?$	GET, GET with query string	

8.2.2 Invalidating Pages

You need to invalidate dynamic pages in the cache when they are no longer valid. To invalidate cached pages, send an XML file with the URL of the pages that you want to invalidate to web cache.

The Employee Benefit application caches the employee information page, which should be invalidated as soon as the data in the database is updated. One way to do this is to send an invalidation message to web cache at the end of the add and remove benefit operations. This method, however, does not invalidate the pages when other applications update the underlying tables in the database that the Employee Benefit application uses.

A better way is to have the database send the invalidation message when the data in the tables changes. To do this, set up triggers on the tables to fire when data in the tables gets updated. The triggers can call a procedure to send the invalidation message to web cache.

The procedure that the triggers invoke looks like the following:

```
-- Usage:
--    SQL> set serveroutput on (When debugging to see dbms_output.put_line’s)
--    SQL> exec invalidate_emp(’doliu-sun’,4001,122);
--
create or replace procedure invalidate_emp (
                            machine in varchar2,
                            port    in integer,
                            emp     in integer) is
  d integer;
  c  utl_tcp.connection;  -- TCP/IP connection to the Web server
  DQUOTE constant varchar2(1) := chr(34);
  CR     constant varchar2(1) := chr(13);
  AMP    constant varchar2(1) := chr(38);
  uri varchar2(100) := ’/empbft/controller?action=queryEmployee’ || AMP ||
                       ’amp;empID=’ || emp;
  content_length integer;
BEGIN
  -- Note: The 177 + Length of uri to invalidate = Content-Length
  content_length := LENGTH(uri) + 177;
  dbms_output.put_line(’Content-Length:’ || content_length);
  --
  -- open connection
  c := utl_tcp.open_connection(machine, port);
  --
  -- Send the HTP Protocol Header
  -- send HTTP POST for Oracle Web Cache
  d := utl_tcp.write_line(c, ’POST /x-oracle-cache-invalidate HTTP/1.0’);
  --
  -- Note: The Authorization passes the User:Password as a base64 encoded
  --       string. ie. invalidator:admin =>
  d := utl_tcp.write_line(c, ’Authorization: BASIC aW52YWxpZGF0b3I6YWRtaW4=’);
  d := utl_tcp.write_line(c, ’Content-Length: ’ || content_length);
  dbms_output.put_line(’Content-Length: ’ || content_length);
  --
  -- send TWO CR’s per HTTP Protocol  (Note: One from above)
  -- (Note: If testing with telnet count cr as 2 characters)
  d := utl_tcp.write_line(c, CR );
  --
  -- send Calypso xml Invalidation File
  d := utl_tcp.write_line(c, '<?xml version=' || DQUOTE || '1.0' || DQUOTE ||
                             '?>');
  d := utl_tcp.write_line(c, '<!DOCTYPE INVALIDATION SYSTEM ' || DQUOTE ||
                             'internal:///invalidation.dtd' || DQUOTE || '>');
  d := utl_tcp.write_line(c, '<INVALIDATION>');
  --
  -- May need to uncomment this for testing dif. expressions.
  -- d := utl_tcp.write_line(c, '<URL EXP=' || DQUOTE || '/cache.htm' || DQUOTE
  --                  || ' PREFIX=' || DQUOTE || 'NO' || DQUOTE || '>');
  d := utl_tcp.write_line(c, '<URL EXP=' || DQUOTE || uri || DQUOTE ||
                             ' PREFIX=' || DQUOTE || 'NO' || DQUOTE || '>');
  --
  d := utl_tcp.write_line(c, '<VALIDITY LEVEL=' || DQUOTE || '0' || DQUOTE ||
                             ' />');
  d := utl_tcp.write_line(c, '</URL>');
  d := utl_tcp.write_line(c, '</INVALIDATION>');
  --
  BEGIN
  LOOP
    -- Capture some of the expected return output when debugging
    dbms_output.put_line(substr(utl_tcp.get_line(c, TRUE),1,80)); -- read result
    dbms_output.put_line(substr(utl_tcp.get_line(c, TRUE),81,160));
    dbms_output.put_line(substr(utl_tcp.get_line(c, TRUE),161,240));
    dbms_output.put_line(substr(utl_tcp.get_line(c, TRUE),241,320));
    dbms_output.put_line(substr(utl_tcp.get_line(c, TRUE),321,400));
    dbms_output.put_line(substr(utl_tcp.get_line(c, TRUE),401,480));
  END LOOP;
  EXCEPTION
    WHEN utl_tcp.end_of_input THEN
      NULL; -- end of input
  END;
  utl_tcp.close_connection(c);
END;
```

The invalidate message that the procedure sends to web cache is an XML file. The file looks like the following:

```
<?xml version="1.0"?>
<!DOCTYPE INVALIDATION SYSTEM "internal:///invalidation.dtd">
<INVALIDATION>
<URL EXP="uri" PREFIX="NO">
```

```
<VALIDITY LEVEL="0" />
</URL>
</INVALIDATION>
```

The *uri* is replaced with something like:

```
/empbft/controller?action=queryEmployee&amp;empID=123
```

Web cache listens on a specific port. The procedure calls **utl_tcp.open_connection** to open a connection to web cache and sends an HTTP header:

```
POST /x-oracle-cache-invalidate HTTP/1.0
Authorization: BASIC aW52YWxpZGF0b3I6YWRtaW4=
Content-Length: contentLength
```

Note that the procedure has to calculate the content length. It starts with 177, which is the length of the XML file, to which it adds the length of the uri.

The Authorization specifies the username and password for web cache.

8.2.3 Setting up Triggers on the Underlying Tables

The underlying tables in the database have the following triggers. These triggers run the invalidate procedure.

The first trigger is fired when a row is deleted from the **employee_benefit_items** table.

```
CREATE OR REPLACE TRIGGER AFTER_DEL_TRIG
AFTER DELETE on employee_benefit_items
FOR EACH ROW
BEGIN
invalidate_emp('doliu-sun', 4001, :old.EMPLOYEE_ID);
END;
```

The second trigger is fired when a row is inserted or updated in the **employee_benefit_items** table.

```
CREATE OR REPLACE TRIGGER AFTER_INS_UPD_TRIG
AFTER INSERT OR UPDATE on employee_benefit_items
FOR EACH ROW
BEGIN
invalidate_emp('doliu-sun', 4001, :new.EMPLOYEE_ID);
END;
```

9 Running in a Portal Framework

To make the Employee Benefit sample application run within a portal framework, you have to make some changes to the application. The changes that you have to make are in the controller and the action handler objects. You also have to edit the links in the JSP files to make them work. The model layer (that is, the Employee and Benefit EJBs) remains the same.

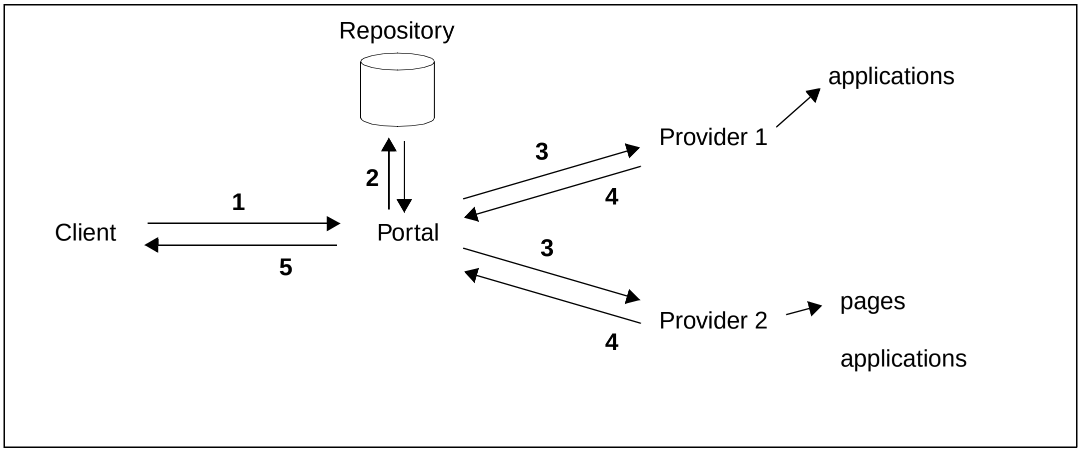
Topics in this chapter:

- Section 9.1, "How Portal Processes Requests"
- Section 9.2, "Screenshots of the Application in a Portal"
- Section 9.3, "Changes You Need to Make to the Application"
- Section 9.4, "Update the Links Between Pages Within a Portlet"
- Section 9.5, "Use include instead of the forward Method"
- Section 9.6, "Protect Parameter Names"
- Section 9.7, "Make All Paths Absolute"

9.1 How Portal Processes Requests

The following figure shows how portal handles requests. This is important in understanding why you have to use APIs in the Java Portal Developer's Kit to code your links and parameters.

Figure 9–1 How portal processes requests



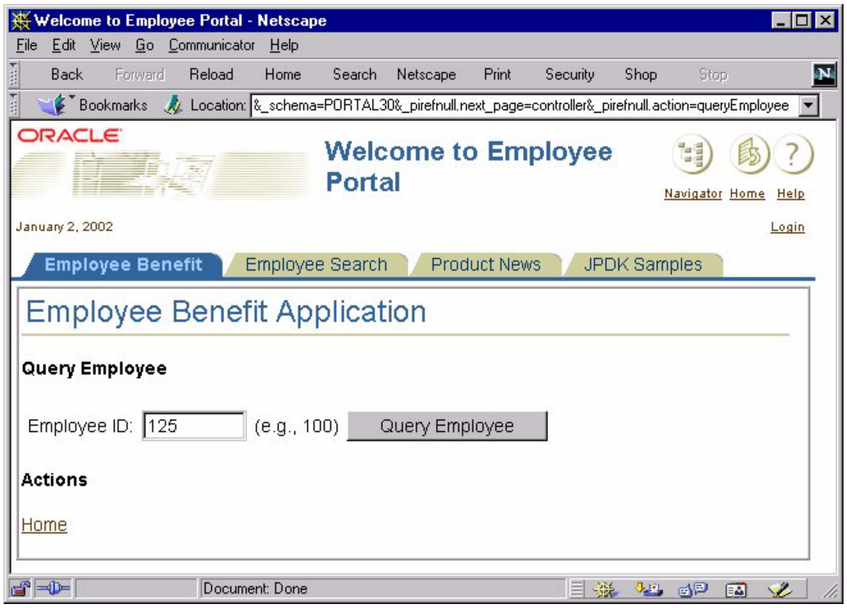
1. A client sends a request to Oracle9*i*AS for a portal page.
2. Portal handles the request. It queries the repository to get a list of portal providers that need to supply data to render the portal page.
3. Portal sends the request to each provider.
4. The providers process the request and return the appropriate data to portal.
5. Portal assembles the data into a page and returns the page to the client.

9.2 Screenshots of the Application in a Portal

The screens for the application in a portal look the same as if the application were running outside of a portal. The only difference is that the portal pages contain tabs and icons as defined by users and administrators. Users and administrators can set up portals with different looks; see the portal documentation for details.

Figure 9–2 to Figure 9–5 show the pages of the application in a portal. You can compare these portal pages with the non-portal pages in Figure 2–1 and Figure 2–2.

Figure 9–2 ID page in a portal


Welcome to Employee Portal - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location:
ORACLE
Navigator Home Help
Login
Employee Benefit
Employee Search
Product News
JPDK Samples
Query Employee
Home
Document: Done

&_schema=PORTAL30&_pirefnull.next_page=controller&_pirefnull.action=queryEmployee

Welcome to Employee Portal

January 2, 2002

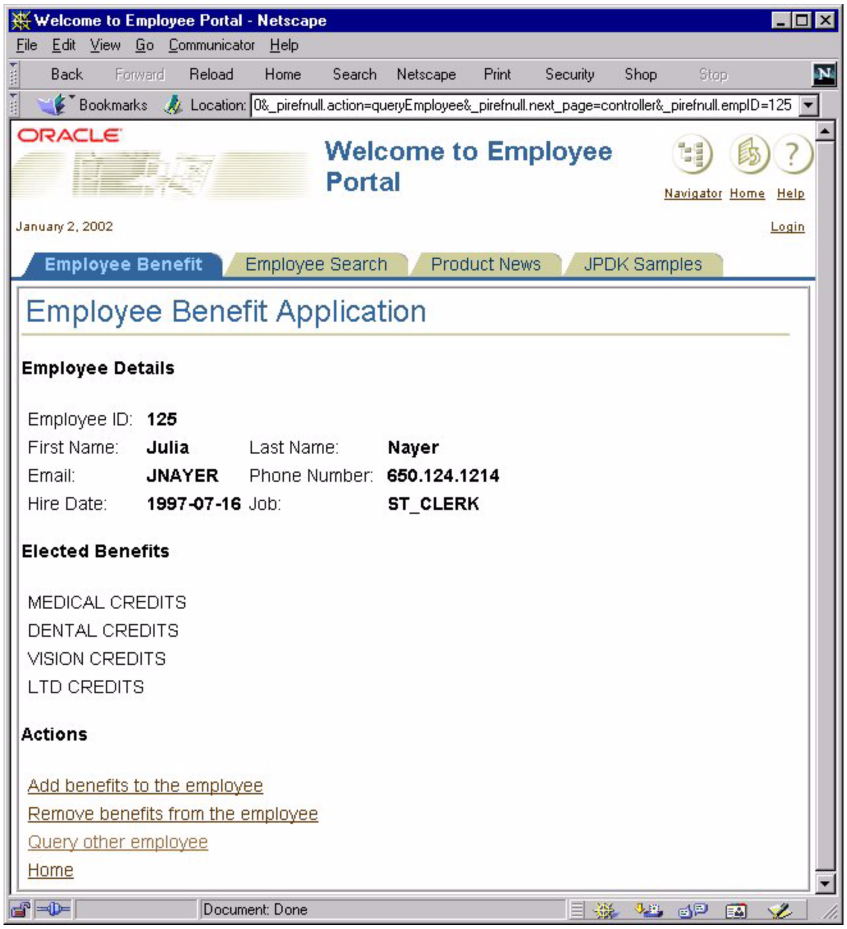
Employee Benefit Application

Query Employee

Employee ID: 125 (e.g., 100) Query Employee

Actions

Figure 9–3 Info page in a portal


Welcome to Employee Portal - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: 0&_pirefnull.action=queryEmployee&_pirefnull.next_page=controller&_pirefnull.empID=125
ORACLE
Welcome to Employee Portal
Navigator Home Help
January 2, 2002
Login
Employee Benefit
Employee Search
Product News
JPDK Samples
Document: Done

Employee Benefit Application

Employee Details

Employee ID:	**125**		
First Name:	**Julia**	Last Name:	**Nayer**
Email:	**JNAYER**	Phone Number:	**650.124.1214**
Hire Date:	**1997-07-16**	Job:	**ST_CLERK**

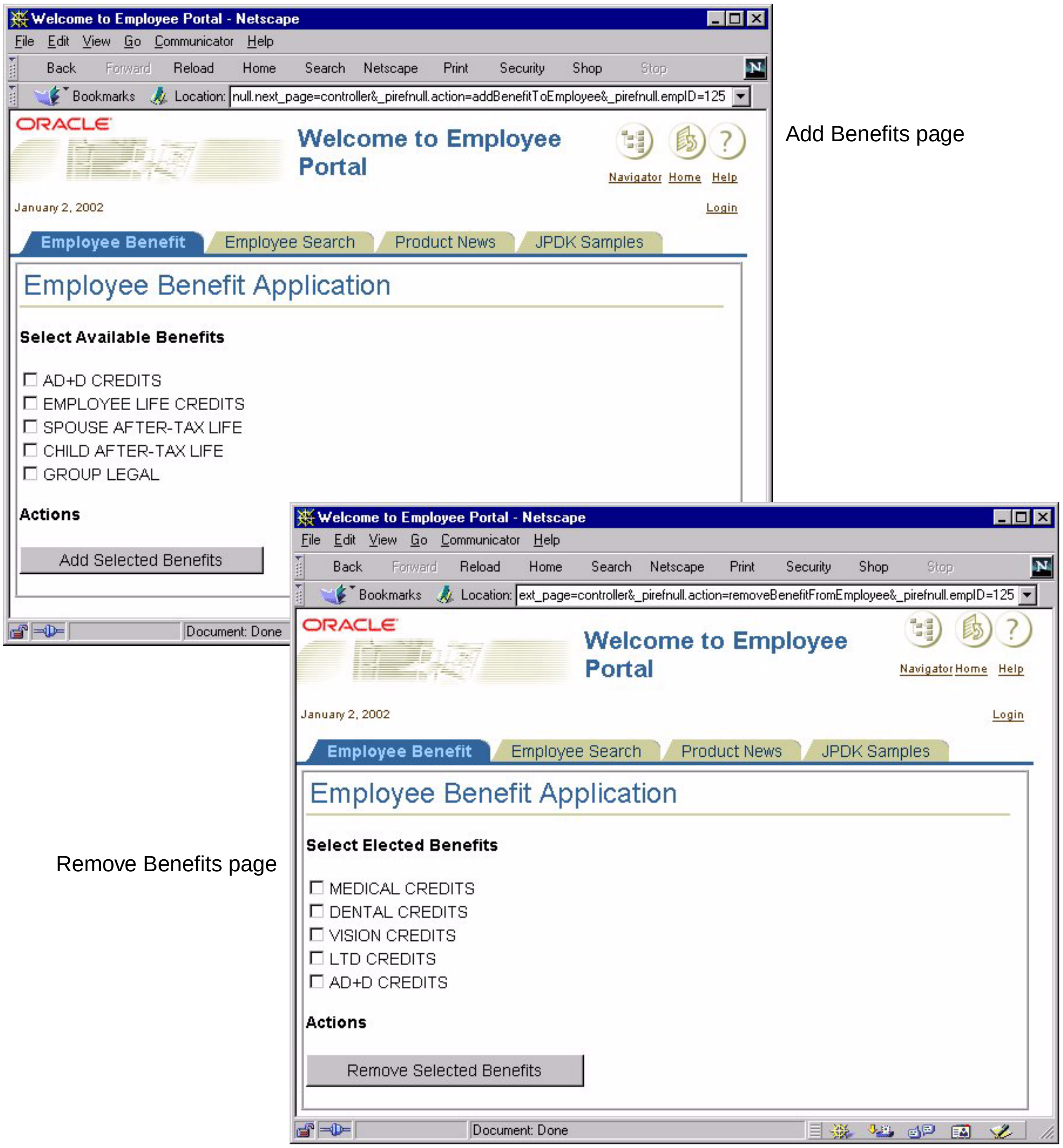
Elected Benefits

MEDICAL CREDITS
DENTAL CREDITS
VISION CREDITS
LTD CREDITS

Actions

Add benefits to the employee
Remove benefits from the employee
Query other employee
Home

Figure 9–4 Add Benefits and Remove Benefits pages in a portal



Location: null.next_page=controller&_pirefnull.action=addBenefitToEmployee&_pirefnull.empID=125

Welcome to Employee Portal

January 2, 2002

Employee Benefit Application

Select Available Benefits

- ☐ AD+D CREDITS
- ☐ EMPLOYEE LIFE CREDITS
- ☐ SPOUSE AFTER-TAX LIFE
- ☐ CHILD AFTER-TAX LIFE
- ☐ GROUP LEGAL

Actions

Add Selected Benefits

Location: ext_page=controller&_pirefnull.action=removeBenefitFromEmployee&_pirefnull.empID=125

Welcome to Employee Portal

January 2, 2002

Employee Benefit Application

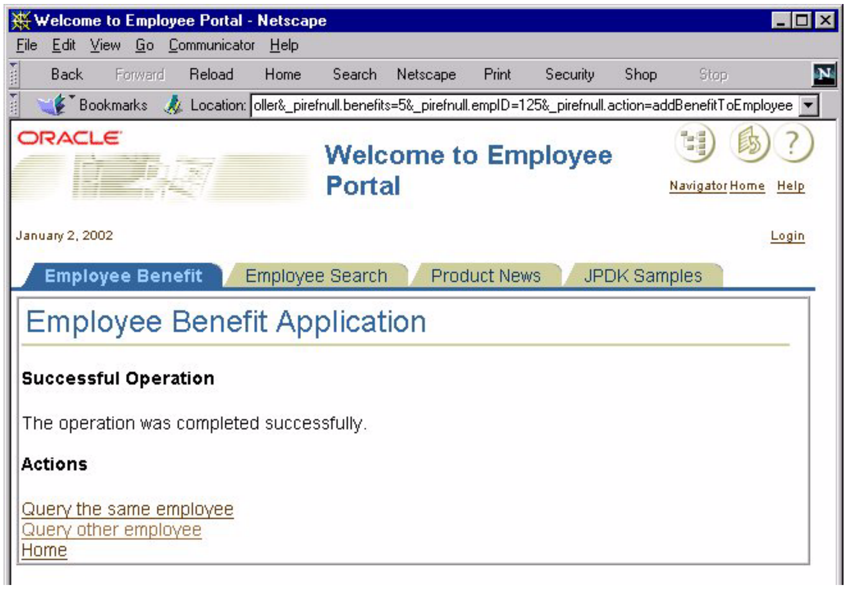
Select Elected Benefits

- ☐ MEDICAL CREDITS
- ☐ DENTAL CREDITS
- ☐ VISION CREDITS
- ☐ LTD CREDITS
- ☐ AD+D CREDITS

Actions

Remove Selected Benefits

Figure 9–5 Success page in a portal


Welcome to Employee Portal - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: oller&_pirefnull.benefits=5&_pirefnull.empID=125&_pirefnull.action=addBenefitToEmployee
ORACLE
Welcome to Employee Portal
Navigator Home Help
January 2, 2002
Login
Employee Benefit
Employee Search
Product News
JPDK Samples

Employee Benefit Application

Successful Operation

The operation was completed successfully.

Actions

Query the same employee
Query other employee
Home

9.3 Changes You Need to Make to the Application

Before you can run the sample application in a portal, you have to set up a few things outside the application as well as make some changes to the application itself.

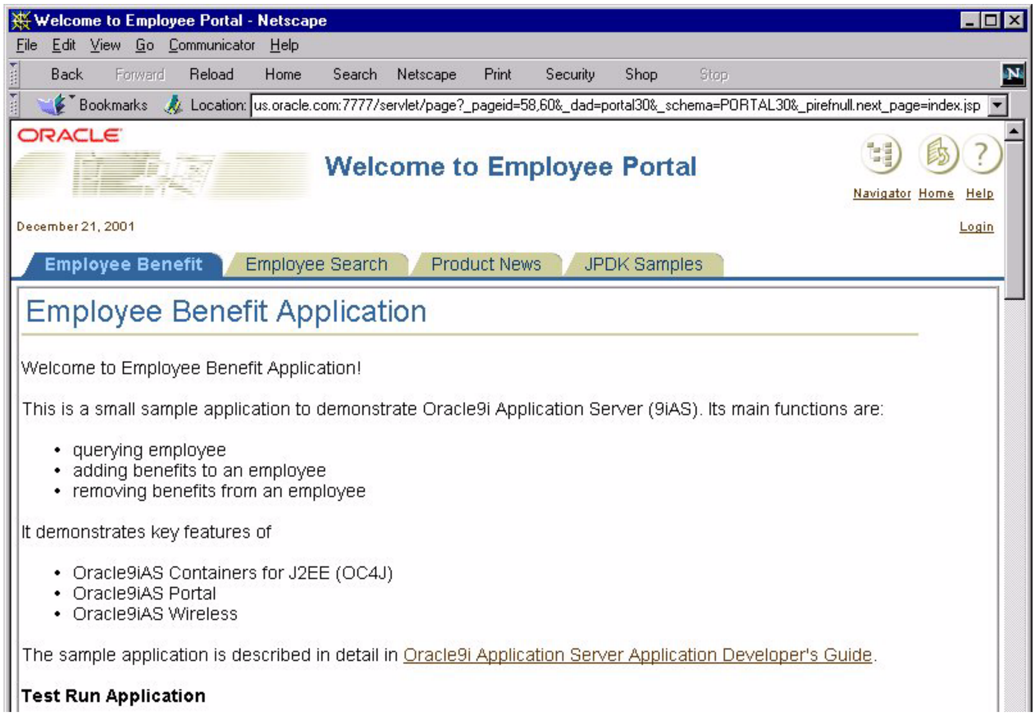
9.3.1 Set up a Provider and a Portal Page

You need to have a portal environment in which to run the application:

- Set up a provider, and register the sample application with the provider.
- Set up a portal page and define one of the regions on the page to display the sample application.

The following figure shows a sample portal page that contains the Employee Benefit application. The tabs at the top of the page take you to different pages in the portal. You can have different tabs in your portal page.

Figure 9–6 A portal page containing the sample application


Welcome to Employee Portal - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: us.oracle.com:7777/servlet/page?_pageid=58,60&_dad=portal30&_schema=PORTAL30&_pirefnull.next_page=index.jsp
ORACLE
Welcome to Employee Portal
Navigator Home Help
December 21, 2001
Login
Employee Benefit
Employee Search
Product News
JPDK Samples

Employee Benefit Application

Welcome to Employee Benefit Application!

This is a small sample application to demonstrate Oracle9i Application Server (9iAS). Its main functions are:

- querying employee
- adding benefits to an employee
- removing benefits from an employee

It demonstrates key features of

- Oracle9iAS Containers for J2EE (OC4J)
- Oracle9iAS Portal
- Oracle9iAS Wireless

The sample application is described in detail in Oracle9i Application Server Application Developer's Guide.

Test Run Application

9.3.2 Edit the Application

You need to add some calls to the JPDK API to make your application run in a portal environment.

- Update the links where you want to display another page within the portlet. If the file that contains the URL is an HTML page, you have to change it to a JSP page because you need to determine the URL dynamically.

  See Section 9.4, "Update the Links Between Pages Within a Portlet".

- Invoke the **include** method instead of **forward**. You have to use **include** because the portal needs to add data from other portlets. If you use **forward**, the portal does not have a chance to gather data from the other portlets.

  See Section 9.5, "Use include instead of the forward Method".

- Use the **portletParameter** method in the **HttpPortletRendererUtil** class to ensure that request parameters have unique names. This ensures that applications on the portal page read only their parameters and not parameters for other applications. This also enables applications to use the same parameter name; the method prefixes parameter names with a unique string for each application.

  See Section 9.6, "Protect Parameter Names".

- Make all URL paths absolute paths using the **absoluteLink** or the **htmlFormActionLink** method in the **HttpPortalRendererUtil** class, depending on the HTML tag.

  See Section 9.7, "Make All Paths Absolute".

9.4 Update the Links Between Pages Within a Portlet

When you need to link from one page in your application to another page within a portlet, you cannot simply specify the target page's URL in the `href` attribute of an `<a>` tag. Instead you have to do the following:

- Use the **parameterizeLink** method in the **HttpPortletRendererUtil** class. See Section 9.4.1, "The parameterizeLink Method".
- Add the **next_page** parameter to the request's query string to specify the target page or object. See Section 9.4.2, "The next_page Parameter".

9.4.1 The parameterizeLink Method

The **parameterizeLink** method enables you to add a query string to the link. (If you do not have a query string in your link, you can just use the **absoluteLink** method. See Section 9.7, "Make All Paths Absolute".)

In the Employee Benefit application, some of the places where you have to use the **parameterizeLink** method are:

- to navigate from the ID page to the Info page
- to navigate from the Info page to the Add Benefit or the Remove Benefit pages

The following files are affected: **addBenefitToEmployee.jsp**, **removeBenefitFromEmployee.jsp**, **queryEmployee.jsp**, **error.jsp**, and **success.jsp**.

The following example shows a link with two parameters in the query string.

- Running outside a portal environment:

```
// from addBenefitsToEmployees.jsp
<a href="/empbft/controller?action=queryEmployee&amp;empID=<%=empId%>">Query
      the same employee</a>
```

- Running within a portal environment:

```
// from addBenefitsToEmployees.jsp
<%
  String fAction = HttpPortletRendererUtil.portletParameter(request,
                        SessionHelper.ACTION_PARAMETER);
  String fEmpId  = HttpPortletRendererUtil.portletParameter(request,
                        SessionHelper.EMP_ID_PARAMETER);
%>
...
<a href="<%=HttpPortletRendererUtil.parameterizeLink(request,
PortletRendererUtil.PAGE_LINK,
HttpPortletRendererUtil.portletParameter(request, "next_page") +
          "=controller" + "&amp;" +
          fAction + "=queryEmployee" + "&amp;" +
          fEmpId + "=" + empId)%>">Query the same employee</a>
```

9.4.2 The next_page Parameter

In the example above, you may have noticed that the target of the link, which is the controller, is specified as the value of the **next_page** parameter. The reason for this is that requests in a portal environment are always directed to the portal. The portal then forwards the requests to providers (see Section 9.1, "How Portal Processes Requests"). For the provider to send the request to a specific target, you specify the target in the **next_page** parameter.

The name of the **next_page** parameter is specified in the **provider.xml** file (in the **WEB-INF/providers/empbft** directory in the **webapp.war** file). You can define the name of the parameter to be anything you want: it is the value of the **pageParameterName** tag.

In URLs for the application, the query string contains the **next_page** parameter. Portal sends the query string to the provider, which does the following:

1. The provider sees **next_page** as a special parameter.
2. The provider sends the request to the value of the parameter (controller).
3. The controller and other objects in the application process the request as normal.

```
// provider.xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<?providerDefinition version="3.1"?>

<provider class="oracle.portal.provider.v2.DefaultProviderDefinition">
   <session>false</session>
   <useOldStyleHeaders>false</useOldStyleHeaders>

   <portlet class="oracle.portal.provider.v2.DefaultPortletDefinition">
      <id>1</id>
      <name>EmployeeBenefit</name>
      <title>Employee Benefit Portlet</title>
      <description>This portlet provides access to Employee Benefit
Application.</description>
      <timeout>10000</timeout>
      <timeoutMessage>Employee Benefit Portlet timed out</timeoutMessage>
      <showEdit>false</showEdit>
      <showEditDefault>false</showEditDefault>
      <showPreview>false</showPreview>
      <showDetails>false</showDetails>
      <hasHelp>false</hasHelp>
      <hasAbout>false</hasAbout>
      <acceptContentType>text/html</acceptContentType>
      <renderer class="oracle.portal.provider.v2.render.RenderManager">
         <renderContainer>true</renderContainer>
         <contentType>text/html</contentType>
         <showPage>index.jsp</showPage>
         <pageParameterName>next_page</pageParameterName>
      </renderer>
   </portlet>

</provider>
```

9.4.3 Linking to the ID Page

This is a special case to link to the ID page of the application. You can use this link as the first link to the application.

To create a link to the ID page and display it in the portal page, you need to use the following URL:

```
<%@ page import="oracle.portal.provider.v2.render.http.HttpPortletRendererUtil" %>
<%@ page import="oracle.portal.provider.v2.render.PortletRendererUtil" %>
...
<a href="<%=HttpPortletRendererUtil.parameterizeLink(request,
     PortletRendererUtil.PAGE_LINK,
     HttpPortletRendererUtil.portletParameter(request, "next_page") + "=controller")%>">
```

Note that the `<a>` tag uses JSP scriptlets. This means that this link has to be in a JSP file; it cannot be in an HTML file.

The `href` attribute uses JPDK APIs to ensure that the portal processes the link correctly, and that the Employee Benefit application sends the request to the controller object. This chapter explains why you have to express the link in this fashion.

After Oracle9*i*AS runs the JSP scriptlet, you end up with a link that looks something like:

```
http://<host>/servlet/page?_pageid=58,60&_dad=portal30&_schema=PORTAL30&
                     _pirefnull.next_page=controller
```

9.5 Use include instead of the forward Method

Call the **include** method instead of **forward**. You have to use **include** because the portal needs to add data from other providers. If you use **forward**, the portal does not have a chance to gather data from the other providers. See Figure 9–1.

- Running outside a portal environment:

  ```
  RequestDispatcher rq = req.getRequestDispatcher(forward);
  rq.forward(req, res);
  ```

- Running within a portal environment:

  ```
  RequestDispatcher rq = req.getRequestDispatcher(forward);
  rq.include(req, res);
  ```

The only class that calls **forward** is the **AbstractActionHandler** abstract class.

9.6 Protect Parameter Names

Ensure that parameter names on your page do not conflict with parameter names from other pages in the portal. To protect your parameters, call the **portletParameter** method in the **HttpPortletRendererUtil** class to ensure that your parameters have unique names. The method prefixes parameter names with a unique string for each application; this enables applications to use the same parameter name safely.

By using the method, you ensure that your applications on the portal page read only their parameters and not parameters from other applications.

You have to use the method to protect all your field names in your HTML forms. You have to do this when retrieving and setting values for the fields.

The following files are affected: **AddBenefitToEmployee.java**, **Controller.java**, **QueryEmployee.java**, **RemoveBenefitFromEmployee.java**, **addBenefitToEmployee.jsp**, **removeBenefitFromEmployee.jsp**, **queryEmployee.jsp**, **error.jsp**, and **success.jsp**.

When you use the methods to protect the parameters, the links look something like the following:

- For add benefit actions:

  ```
  http://<host>/servlet/page?_pageid=58%2C60&_dad=portal30&_schema=PORTAL30&
  _pirefnull.action=addBenefitToEmployee&_pirefnull.next_page=controller&
  _pirefnull.empID=125
  ```

- For remove benefit actions:

  ```
  http://<host>/servlet/page?_pageid=58%2C60&_dad=portal30&_schema=PORTAL30&
  _pirefnull.action=removeBenefitFromEmployee&
  _pirefnull.next_page=controller&_pirefnull.empID=125
  ```

- For query employee actions:

  ```
  http://<host>/servlet/page?_pageid=58%2C60&_dad=portal30&_schema=PORTAL30&
  _pirefnull.action=queryEmployee&_pirefnull.next_page=controller&
  _pirefnull.empID=125
  ```

The parameters used by the application are prefixed with **_pirefnull**. The other parameters in the URL are required by portal. Note also that the URL does not point to the controller directly. Instead it uses the **_pirefnull.next_page** parameter to indicate that the controller should handle the request. See Section 9.4.2, "The next_page Parameter" for details.

9.6.1 Retrieving Values

The following example retrieves the values of two parameters.

- Running outside a portal environment:

```
// from AddBenefitToEmployee.java
String benefits[] = req.getParameterValues(SessionHelper.BENEFIT_PARAMETER);
String client = req.getParameter(SessionHelper.CLIENT_TYPE_PARAMETER);
```

- Running within a portal environment:

```
// from AddBenefitToEmployee.java
import oracle.portal.provider.v2.render.http.HttpPortletRendererUtil;
...
String fBenefits = HttpPortletRendererUtil.portletParameter(req,
                   SessionHelper.BENEFIT_PARAMETER);
String benefits[] = req.getParameterValues(fBenefits);
String fClient = HttpPortletRendererUtil.portletParameter(req,
                 SessionHelper.CLIENT_TYPE_PARAMETER);
String client = req.getParameter(fClient);
```

9.6.2 Setting Values

If your parameter is a form element (for example, a checkbox or a hidden element), you have to call the **portletParameter** method to protect the name before you can use it. The following example shows how to set the BENEFIT_PARAMETER in a form:

```
// from addBenefitsToEmployees.jsp
String fBenefits = HttpPortletRendererUtil.portletParameter(
                       request, SessionHelper.BENEFIT_PARAMETER);
<form ... >
...
<input type="checkbox" name="<%=fBenefits%>" value="<%=b.getId()%>">
```

9.7 Make All Paths Absolute

Make all URL paths absolute paths using the **absoluteLink** or the **htmlFormActionLink** method in the **HttpPortalRendererUtil** class, depending on the HTML tag.

You cannot use paths relative to the current page because Oracle9*i*AS sends requests to portal first, and portal sends requests to providers. See Figure 9–1. When providers get the requests, the current path is the portal, not to the current page. By using absolute paths, you ensure that the provider can find the proper object.

The following files are affected: **addBenefitToEmployee.jsp**, **removeBenefitFromEmployee.jsp**, **queryEmployee.jsp**, **error.jsp**, and **success.jsp**.

9.7.1 <a> and <link> Tags

Use the **absoluteLink** method to qualify paths in `<a>` and `<link>` tags.

- Running outside a portal environment:

```
// from addBenefitToEmployee.jsp
<link rel="stylesheet" type="text/css" href="css/blaf.css">
```

- Running within a portal environment:

```
// from addBenefitToEmployee.jsp
<link rel="stylesheet" type="text/css"
  href="<%= HttpPortletRendererUtil.absoluteLink(request,
                       "./css/blaf.css")%>"
>
```

9.7.2 <form> Tag

Use the **htmlFormActionLink** method to qualify paths in the <form> tag.

- Running outside a portal environment:

```
// from addBenefitToEmployee.jsp
<form method="GET" action="/empbft/controller">
```

- Running within a portal environment:

```
// from addBenefitToEmployee.jsp
<form method="GET"
  action="<%=HttpPortletRendererUtil.htmlFormActionLink(request,
                            PortletRendererUtil.PAGE_LINK)%>">
```

```
<%=HttpPortletRendererUtil.htmlFormHiddenFields(request,
          PortletRendererUtil.PAGE_LINK)%>
<input type="hidden"
  name="<%=HttpPortletRendererUtil.portletParameter(request, "next_page")%>"
  value="controller">
```

Note that in the portal version the action attribute does not point to the controller. Instead, it points to the portal. The actual target for the form is specified in a hidden field called **next_page**. The value of the hidden field specifies the target. See Section 9.4.2, "The next_page Parameter" for details.

When you use forms, you need to include additional parameters such as `_dad` and `_schema`. These parameters are needed by portal. To include these parameters, you can use the **htmlFormHiddenFields** method.

10 Enhancements to the Application

You can make the application even more useful by making the following additions:

- Section 10.1, "Adding Security Features"
- Section 10.2, "Publishing the Application as a Web Service"

10.1 Adding Security Features

As it stands, any user can enter an employee ID and make changes to that employee’s benefits. This is probably not what you want. You can add a login page that prompts the user to enter a login ID and password. The application can then verify the login against a directory server.

10.2 Publishing the Application as a Web Service

Web services enable diverse clients, independent of language or operating system, to invoke methods on your application. In the Employee Benefit application, you can expose business logic methods by exposing the **EmployeeManager** session bean. Typically, you would expose methods declared in the session bean’s remote interface. Clients can then process the return values as they see fit.

For details on web services, see the *Oracle9iAS Web Services Developer’s Guide*.

A
Configuration Files

This appendix shows the configuration files needed to deploy and run the Employee Benefit application:

- Section A.1, "server.xml"
- Section A.2, "default-web-site.xml"
- Section A.3, "data-sources.xml"

The **server.xml** and **default-web-site.xml** files define the Employee Benefit application (**empbft**). They also define **omsdk**, which is an application that the wireless feature in Oracle9*i*AS uses.

These configuration files are located in the **$J2EE_HOME/config** directory.

For a detailed description of configuration files, see the *Oracle9i*AS *User's Guide*.

A.1 server.xml

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE application-server PUBLIC "Orion Application Server Config"
                 "http://xmlns.oracle.com/ias/dtds/application-server.dtd">
<application-server
       application-directory="../applications"
       deployment-directory="../application-deployments"
       transaction-log="../persistence/omsdk.state" >
   <rmi-config path="./omsdk-rmi.xml" />
   <log>
      <file path="../log/omsdk-server.log" />
   </log>
   <global-application name="default" path="./application.xml" />
   <global-web-app-config path="global-web-application.xml" />
   <web-site path="./omsdk-web-site.xml" />
   <application name="omsdk"
             path="../applications/omsdk.ear" auto-start="true" />
   <application name="empbft"
             path="../applications/empbft.ear" auto-start="true" />
</application-server>
```

A.2 default-web-site.xml

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE web-site PUBLIC "Oracle9iAS XML Web-site"
                 "http://xmlns.oracle.com/ias/dtds/web-site.dtd">
<!-- change the host name below to your own host name. Localhost will -->
<!-- not work with clustering -->
<web-site
       port="9000"
       display-name="Default Oracle9iAS Containers for J2EE Web Site">
   <default-web-app application="default" name="defaultWebApp" />
   <web-app application="omsdk" name="omsdk" root="/omsdk"
                   load-on-startup="true"/>
   <web-app application="empbft" name="web" root="/empbft"
                   load-on-startup="true"/>
   <!-- Access Log, where requests are logged to -->
   <access-log path="../log/omsdk-web-access.log" />
</web-site>
```

A.3 data-sources.xml

```
<?xml version="1.0"?>
<!DOCTYPE data-sources PUBLIC "Orion data-sources"
               "http://xmlns.oracle.com/ias/dtds/data-sources.dtd">
<data-sources>
   <data-source
       class="com.evermind.sql.DriverManagerDataSource"
       name="OracleDS"
       location="jdbc/OracleCoreDS"
       xa-location="jdbc/xa/OracleXADS"
       ejb-location="jdbc/OracleDS"
       connection-driver="oracle.jdbc.driver.OracleDriver"
       username="hr"
       password="hr"
       url="jdbc:oracle:thin:@doliu-sun:1521:db3"
       inactivity-timeout="30"
   />
</data-sources>
```

Index

A

absolute paths (for portal), 9-15
absoluteLink method (for portals), 9-8, 9-9, 9-15
AbstractActionHandler abstract class, 4-8, 9-12
action handlers, 4-8
  wireless support and, 7-2
Action tag (in wireless clients), 7-16
ActionHandler interface, 4-8
ActionHandlerFactory class, 4-8
add benefits, 6-9, 6-10
  for wireless clients, 7-16
  retrieving benefits for a user, 6-11
  sequence (high-level), 6-10
  sequence diagram, 6-13
  updating database, 6-12
Add Benefits page, 2-5
  for portal, 9-5
  for wireless, 7-4
Add More Benefits page (wireless), 7-5
AddBenefitToEmployee class, 3-6, 7-17
addBenefitToEmployee.jsp, 4-21
addBenefitToEmployeeWireless.jsp, 7-17
AppJNDINames class, 4-25

B

benefit data, retrieving, 6-7
BenefitCatalog bean, 4-18
  details, 4-20
  home interface, 4-18
  remote interface, 4-19
BenefitItem class, 4-20
BenefitModel class, 4-19, 4-20
benefits table, 2-7
blueprints URL on Sun site, 3-7
business logic, 3-6
  objects needed for, 4-2

C

caching. *See* web cache.
chaining pages design, 3-3
clientType parameter, 7-7, 7-11
configuration files, A-1
  data-sources.xml, A-2
  default-web-site.xml, A-2
  server.xml, A-1
content type, for wireless clients, 7-12
controller, 3-6, 4-5
  mappings to classes, 3-6

D

DAOs, 3-7
  employee data and, 4-14
  implementation, 4-14
  interface for, 4-14
  JDBC, 4-15
  specifications for, 4-14
  SQLJ, 4-15
data access objects. *See* DAOs.
database schema, 2-7
data-sources.xml, A-2
Debug class, 4-25
default-web-site.xml, A-2
design of application, 3-1
  chaining pages, 3-3

design goals, 3-2
model-view-controller, 3-4
Design Patterns Catalog URL on Sun site, 3-7
development steps, 1-5
development tools, 1-6

E

EJBs, 1-2
employee data, 4-9
needed by application, 4-2
when to use, 4-3
empbft.xml, 7-4, 7-7, 7-11
Employee bean, 4-9
DAO for, 4-14
home interface, 4-10
load method, 4-12
persistence, 4-12
remote interface, 4-11
Employee Benefit application
accessing from desktop browsers, 6-3
accessing from wireless clients, 7-18
add benefit operation (high-level), 6-10
caching, 8-1
clientType parameter, 7-7
configuration files, A-1
design, 3-1
development steps, 1-5
development tools, 1-6
differences between wireless and desktop versions, 7-5
EJBs, 4-2
objects needed, 4-2
overview, 1-1
portal support, 9-1
query employee operation (high-level), 6-3
remove benefit operation (high-level), 6-14
requirements, 2-2
screenshots (desktop version), 2-3
screenshots (portal version), 9-3
screenshots (wireless version), 7-3
starting URL (for desktop browsers), 6-3
starting URL (for portal), 9-11
starting URL (for wireless clients), 7-18
updating links in a portal, 9-9
wireless support, 7-1
employee data, 4-9
retrieving, 6-4
employee_benefit_items table, 2-7
EmployeeDAO interface, 4-14
EmployeeDAOImpl class, 4-14, 4-16
EmployeeManager bean, 4-21
home interface, 4-22
JSPs and, 6-2
remote interface, 4-23
EmployeeModel class, 4-12
employees table, 2-7
Enterprise JavaBeans. *See* EJBs.
entity beans
Employee bean, 4-9
JSPs and, 4-21
Error page, 2-6
errorWireless.jsp, 7-17

F

findByPrimaryKey method (Employee bean), 4-17, 6-6
findByPrimaryKey method (EmployeeDAOImpl class), 6-7

H

header information for wireless clients, 7-12
hr schema, 2-7
hr.benefits table, 2-7
hr.employee_benefit_items table, 2-7
hr.employees table, 2-7
htmlFormActionLink method, 9-15
htmlFormActionLink method (for portals), 9-8, 9-15
HttpPortletRendererUtil class, 9-8

I

ID page, 2-4
for portal, 9-3
for wireless, 7-3
IDEs
JDeveloper, 1-6

include method (for portals), 9-8, 9-12
Info page, 2-4
  for portal, 9-4
  for wireless, 7-3
invalidating pages in web cache, 8-2, 8-4
  using database triggers, 8-7

J

J2EE, 1-2
  specifications, 1-2
jar, 1-6
Java Authentication and Authorization Service (JAAS), 1-2
Java Message Service (JMS), 1-2
Java objects
  when to use, 4-3
Java Transaction API (JTA), 1-2
javac, 1-6
JavaServer Pages. *See* JSPs.
JDBC, 4-15
JDeveloper, 1-6
JSPs, 1-2, 5-5
  advantages of, 5-5
  EmployeeManager bean and, 6-2
  entity beans and, 4-21
  supporting different client types, 5-6
  tag libraries, 5-5

L

load method, 4-12, 4-16

M

model (business logic), 3-6
model-view-controller (MVC) design. *See* MVC.
MVC, 3-4
  controller, 3-6, 4-5
  diagram, 3-4
  model, 3-6
  view, 3-7

N

next_page parameter, 9-10

O

objects needed by application, 4-2
Oracle Web Cache Manager, 8-2, 8-4

P

pageParameterName tag, 9-10
parameter names and portals, 9-13
parameterizeLink method (for portals), 9-9
performance
  using web cache to improve, 8-1
persistence
  Employee bean, 4-12
portal support, 9-1
  absolute paths, 9-15
  absoluteLink method, 9-8, 9-15
  changes to the application, 9-7
  htmlFormActionLink method, 9-8, 9-15
  include method, 9-8, 9-12
  next_page parameter, 9-10
  pageParameterName tag, 9-10
  parameter names, 9-13
  parameterizeLink method, 9-9
  portletParameter method, 9-8, 9-13
  provider.xml file, 9-10
  request processing, 9-2
  retrieving parameter values, 9-14
  sample figure, 9-7
  screenshots, 9-3
  setting parameter values, 9-14
  setting up provider, 9-7
  updating links between pages, 9-9
portals, 1-2
portletParameter method (for portals), 9-8, 9-13
presentation data, 3-7
  wireless clients, 7-3, 7-7
provider, portal, 9-7
provider.xml file (for portals), 9-10

Q

query employee, 6-3
  findByPrimaryKey method, 6-6
  for wireless clients, 7-13
  retrieving benefit data, 6-7
  retrieving data, 6-4
  sequence (high-level), 6-3
  sequence diagram, 6-8
Query Employee button, 2-3
QueryEmployee class, 3-6
queryEmployee.jsp, 4-20, 6-4
queryEmployeeWireless.jsp, 7-15

R

references
  blueprints, 3-7
  DAO, 4-14
  Design Patterns Catalog, 3-7
  J2EE specifications, 1-2
remove benefits, 6-9, 6-14
  for wireless clients, 7-16
  getting benefits list, 6-15
  sequence (high-level), 6-14
  sequence diagram, 6-17
  updating database, 6-16
Remove Benefits page, 2-5
  for portal, 9-5
Remove More Benefits page (wireless), 7-5
RemoveBenefitFromEmployee class, 3-6, 7-17
removeBenefitFromEmployee.jsp, 4-21
removeBenefitFromEmployeeWireless.jsp, 7-17
requests
  getting origin of (wireless or desktop), 7-7
requirements of application, 2-2

S

sample application. *See* Employee Benefit application.
schema, database, 2-7
screenshots of application
  desktop browser client, 2-3
  portal version, 9-3
  wireless client, 7-3
sequence diagrams
  add benefits, 6-13
  query employee, 6-8
  remove benefits, 6-17
server.xml, A-1
servlets, 1-2, 5-2
  automatic compilation, 5-2
  calling EJB, 5-3
  example, 5-3
  when to use, 4-3
session beans
  BenefitCatalog bean, 4-18
  EmployeeManager bean, 4-21
SessionHelper class, 4-25
SimpleResult DTD, 7-3
specifications
  J2EE, 1-2
SQLJ, 4-15
Success page, 2-5
  for portal, 9-6
successWireless.jsp, 7-5, 7-17

T

tag libraries for JSPs, 5-5
technologies used, 1-2
  J2EE, 1-2
  portals, 1-2
  wireless support, 1-3
text / vnd.oracle.mobilexml value for contentType, 7-12
triggers to invalidate pages in web cache, 8-7

U

utility classes
  AppJNDINames class, 4-25
  Debug class, 4-25
  SessionHelper class, 4-25

V

view, 3-7

W

web cache, 8-1
- analyzing the application, 8-3
- choosing pages to cache, 8-2
- invalidating pages, 8-2, 8-4, 8-7
- Oracle Web Cache Manager, 8-4
- specifying pages to cache, 8-3

wireless support, 1-3, 7-1
- accessing the application, 7-18
- action handler objects, 7-2
- add benefits operation, 7-16
- changes to the application, 7-2
- clientType parameter, 7-7
- content type, 7-12
- details, 7-13
- differences from desktop application, 7-5
- header information, 7-12
- presentation data, 7-3, 7-7
- query operation, 7-13
- remove benefits, 7-16
- screens, 7-3
- SimpleResult DTD, 7-3
- using a simulator, 7-18
- using actual wireless devices, 7-18