Oracle® Application Server 10*g*

Application Developer's Guide

10*g* (9.0.4)

Part No. B10378-01

September 2003

ORACLE®

Oracle Application Server 10*g* Application Developer's Guide, 10*g* (9.0.4)

Part No. B10378-01

Contents

5 Creating Presentation Pages

6 Tracing Flows Between Clients and Business Logic Objects

7 Configuring OracleAS Web Cache for the Application

8 Supporting Wireless Clients

9 Running in a Portal Framework

Send Us Your Comments

Oracle Application Server 10*g* Application Developer's Guide, 10*g* (9.0.4)

Part No. B10378-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. 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 about this manual?

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

- Electronic mail: appserverdocs_us@oracle.com
- FAX: 650-506-7375 Attn: Oracle Application Server Documentation Director
- Postal service:
  Oracle Corporation
  Oracle Application Server Documentation
  500 Oracle Parkway, M/S 1op6
  Redwood Shores, CA 94065
  USA

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

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

Preface

This 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 Oracle Application Server. You should use this guide along with the *Oracle Application Server Containers for J2EE User's Guide*.

This preface contains these topics:

Intended Audience

This guide is intended for developers who perform the following tasks:

- Design and create J2EE 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 guides:

- *Oracle Application Server 10g Concepts*
- *Oracle Application Server Containers for J2EE User's Guide*
- *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide*
- *Oracle Application Server Containers for J2EE Servlet Developer's Guide*
- *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide*

Conventions

This guide uses the following conventions:

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
Bold	Boldface type in text indicates objects (such as buttons and fields) on screens.
Italics	Italic type indicates book titles or emphasis.
`Monospace`	Monospace type indicates names of files, directories, commands.
`Monospaced italics`	Monospaced italic type indicates placeholders or variables for which you must supply particular values.
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.
|	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.
. . .	Horizontal ellipsis points indicate either: ■ That we have omitted lines of the code that are not related to the example ■ That you can repeat a portion of the code

Part I

Overview of Sample Applications and Contents of Database

This guide describes sample applications that access the same schemas in a database. Part I provides an overview of the applications and the schemas.

Part I contains the following chapters:

1

Creating Applications: Overview

When you create applications to be deployed on Oracle Application Server, you might think of different ways to design them. This guide walks you through the design and implementation of three sample applications, and in the process of doing so, it discusses the available options and the advantages and disadvantages of each option.

The resulting applications are 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 applications enable users to view data such as employee name, phone, email, and job ID. Users can add or remove their benefit elections. The applications retrieve and update data in an Oracle database.

The sample applications use many technologies, including JavaServer Pages, servlets, Enterprise JavaBeans, JDBC, portals, wireless devices, web cache, web services, JNDI, and JAAS.

Contents of this chapter:

1.1 Overview of OracleAS

OracleAS supports many technologies. As a result, you have many choices when you design and create your applications. The following sections describe some of the key technologies.

1.1.1 J2EE

The J2EE support includes:

- Enterprise JavaBeans, which enable 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 can 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`

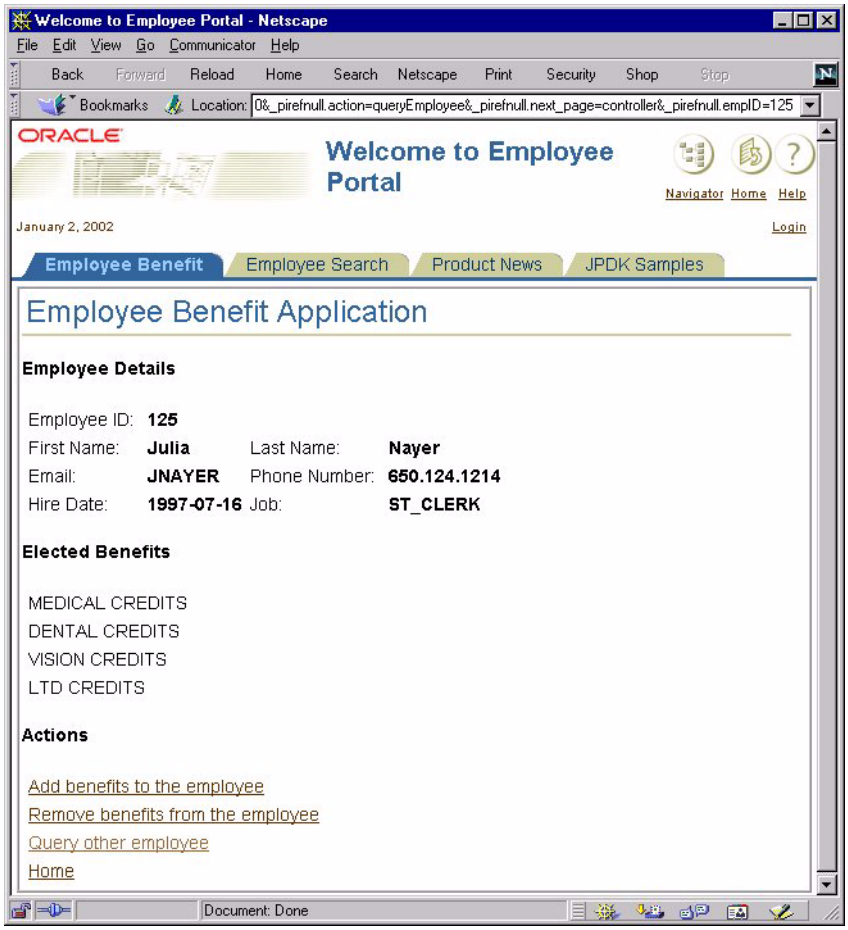
1.1.2 Enterprise Portals

Portals in OracleAS enable you to aggregate, or group, your applications on 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



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

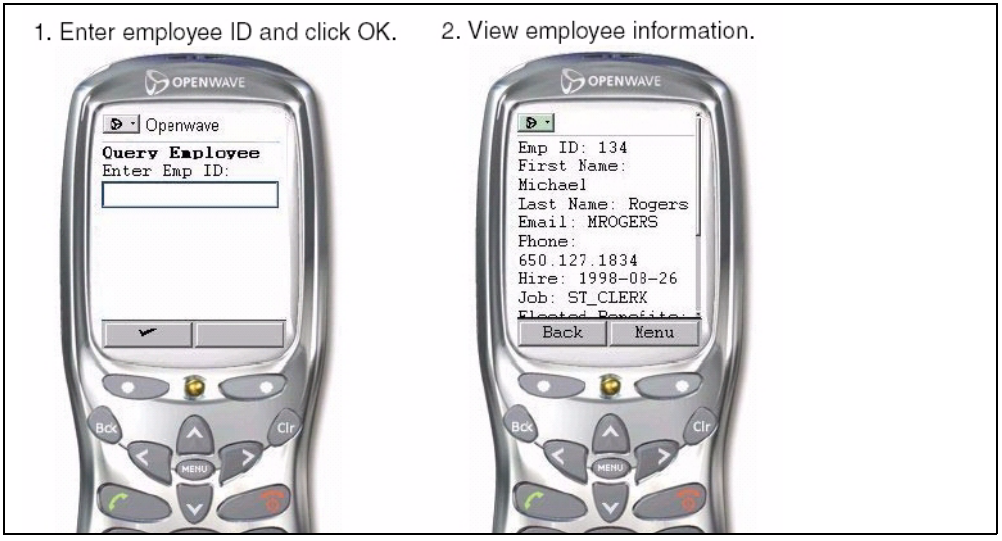
hand, understand different sets of tags and attributes, depending on the wireless device, and speak different protocols.

To make it easy for application developers, the wireless feature in OracleAS 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 a markup language understood 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 first sample application shows how to do that. See Chapter 8, "Supporting Wireless Clients" for details.

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

Figure 1–2 An Application Running on a Cell Phone



1.2 The Sample Applications

This guide describes three sample applications. Table 1–1 lists the operations, technologies, and design patterns associated with each application:

Table 1–1 Comparing the Sample Applications

	First Sample Application	Second Sample Application	Web Services Client Sample Application
Design Pattern Used	Model-View-Controller (MVC)	Session facade	MVC
Supported Operations	Users can perform the following operations: ■ Enter an employee ID and view data such as first name, last name, phone, email, and benefits for the specified employee. ■ Add benefits for the employee. ■ Remove benefits for the employee.	Users can access the following operations through Web Services: ■ List all the existing benefits. ■ Create a new benefit. ■ Get the total number of employees. ■ Enter an employee ID and view the benefits for the specified employee. ■ Enter a benefit ID and get the number of employees enrolled in the specified benefit.	This is a client application that invokes the operations provided by the second application through Web Services.
Technologies Used	■ Servlets ■ JSPs ■ EJBs, including bean-managed persistence, remote interfaces ■ Portal ■ Wireless ■ Web Cache	■ EJB 2.0, including EJB QL, container-managed persistence, container-managed relationships, and local interfaces ■ Web Services	■ Servlets ■ JSPs ■ Web Services

1.3 Database Schema

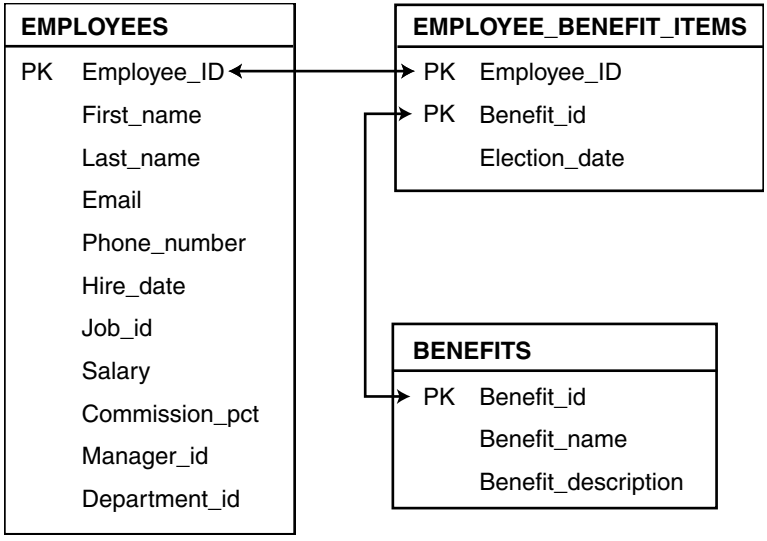
The sample applications use the HR schema that comes with Oracle9*i* database and OracleAS Metadata Repository. The applications use the EMPLOYEES table, plus

two additional tables (BENEFITS and EMPLOYEE_BENEFIT_ITEMS) that you need to install. You install these tables in the default tablespace of the HR schema.

Table 1–2 Tables in the HR Schema

Table	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, benefit_id, and election_date. An employee can have multiple benefits. This is the table that the application updates when employees update their benefit elections.

Figure 1–3 Database Schema



1.4 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 3.1, "Requirements for the First 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.

1.5 Development Tools

To create applications for OracleAS, 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 Oracle JDeveloper 10*g*. Oracle JDeveloper supports each stage in the development lifecycle: it contains UML modelling and generation tools, debugging tools, profiling tools, and tuning tools.

Oracle JDeveloper is closely integrated with OracleAS: you can deploy applications on OracleAS from Oracle JDeveloper.

1.6 What This Guide Covers and Does Not Cover

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

It also shows how to deploy and configure the application.

It does not describe in full the APIs that the application uses. For that information, refer to the *Oracle Application Server Containers for J2EE User's Guide*.

This guide assumes the reader has some concept of servlets, JSPs, portals, web services, wireless devices, and introductory knowledge of EJBs. For more information on these topics, see the OracleAS Documentation Library, available on the Documentation CD-ROM. The library is also available on Oracle Technology Network (`http://otn.oracle.com`).

To read the Java specifications, see:

`http://java.sun.com`

2

Designing the Application

There are several ways to design the sample applications. 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. Yet another way is to use session facade design pattern.

Contents of this chapter:

The session facade design pattern is described in Chapter 10, "Updating EJBs to Use EJB 2.0 Features".

2.1 Design Goals

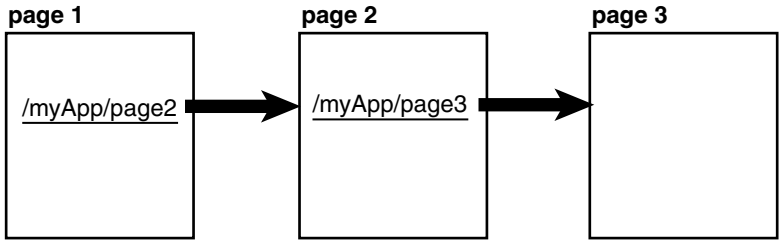
You want to design your application such that changes to one part of the application has minimal or no impact on other parts. This enables you to:

- 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

2.2 Chaining Pages

In the chaining pages design, pages in an 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 2–1 Chaining Pages



Each page can be generated differently. For example, 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 on 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 hard to maintain 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.

2.3 Using Model-View-Controller (MVC)

A better way of designing an 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 way. 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/blueprints/patterns/MVC.html
```

Which Sample Applications Use MVC

The first and the Web Services client sample applications use MVC. The second application uses the session facade design pattern.

For a description of MVC in the first application, see the chapters in Part II, "The First Sample Application".

For MVC in the Web Services client application, see Chapter 11, "Enabling Web Services in the Application".

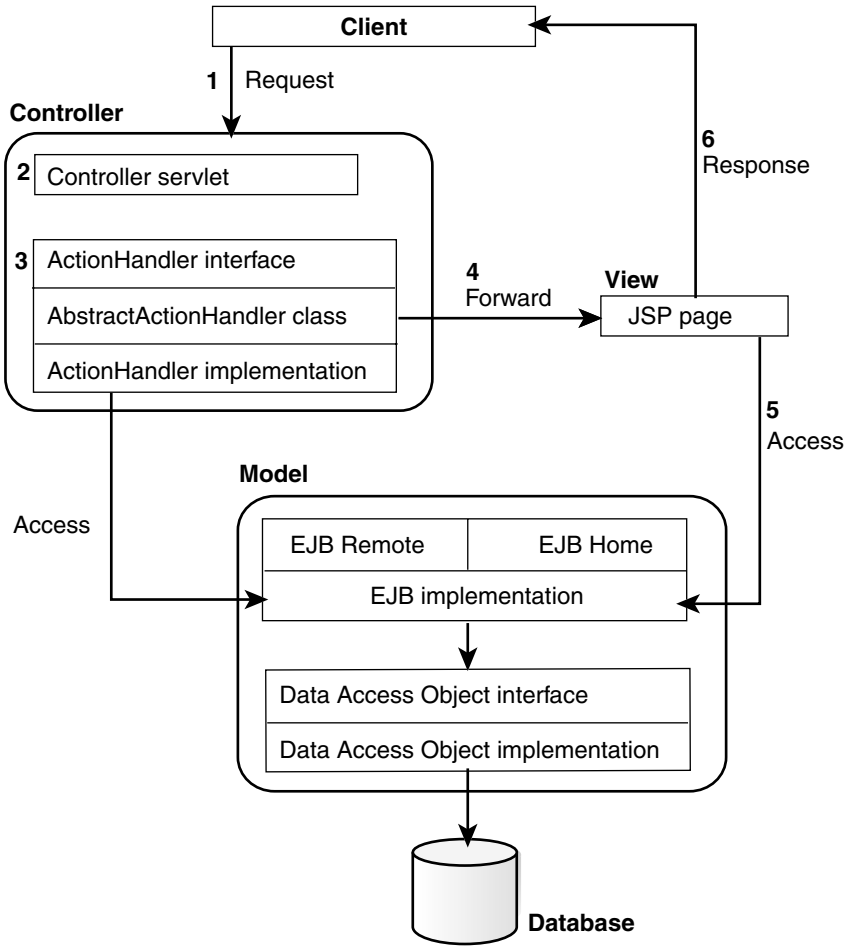
2.3.1 MVC Diagram

Figure 2–2 figure shows a high-level structure of an MVC application. When it 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 2–2 Application Architecture



2.3.2 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 to a class that handles the request. For example, the first sample application has the following mappings:

Table 2–1 Mappings in the Controller for the First 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. 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 mappings 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.

In the first application, the mappings are hardcoded.

In the Web Services client application (which invokes the operations in the second application through Web Services), the mappings are specified as initialization parameters for the controller servlet. The second application does not use the MVC design pattern.

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

Using the controller 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.

2.3.3 Model (Business Logic)

The model refers to 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 first 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.

Database Access

Typically, objects in the model read and update data in databases. If you are using Enterprise JavaBeans objects to access the database, you have these options:

- Use entity beans with bean-managed persistence (BMP). For BMP entity beans, you have to write your own code to read and update data in the database.
- Use entity beans with container-managed persistence (CMP). For CMP entity beans, you configure the beans in the deployment descriptors.

BMP entity beans: Consider using DAO (data access object) to separate the database access portion of your application from the BMP entity beans. The BMP entity beans invoke methods in the DAO classes to access the database. 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 first sample application uses BMP entity beans and DAOs. 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/blueprints
http://java.sun.com/blueprints/patterns/catalog.html
```

CMP entity beans: If you want the EJB container to manage database access for you, you can use CMP entity beans. In the deployment descriptor files, you map the

entity beans to tables in the database. The container associates fields in the entity bean with columns in the table.

You can also define container-managed relationships (CMRs) between entity beans in the deployment descriptor files. CMRs enable you to retrieve data for a specific entity bean based on data from another entity bean.

The second sample application uses CMP entity beans and CMRs. See Chapter 10, "Updating EJBs to Use EJB 2.0 Features".

2.3.4 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 applications, 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 applications, all the presentation code is in JSP files. The JSP files call on EJBs and servlets to process requests.

Part II

The First Sample Application

This part of the guide describes the first sample application. It contains the following chapters:

- Chapter 3, "The First Sample Application: Requirements and Screenshots"
- Chapter 4, "Implementing Business Logic"
- Chapter 5, "Creating Presentation Pages"
- Chapter 6, "Tracing Flows Between Clients and Business Logic Objects"
- Chapter 7, "Configuring OracleAS Web Cache for the Application"
- Chapter 8, "Supporting Wireless Clients"
- Chapter 9, "Running in a Portal Framework"

3

The First Sample Application: Requirements and Screenshots

This chapter describes the requirements and the screens in the first sample application.

Contents of this chapter:

3.1 Requirements for the First Sample Application

The 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 3–1). See Section 1.3, "Database Schema" for table details.
- 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)

3.2 Screenshots of the First Sample Application

When the user invokes the application, the first page prompts the user to enter an employee ID (Figure 3–1).

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 3–1).

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

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 3–2). 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 screenshots of the application running on a wireless device, see Chapter 8, "Supporting Wireless Clients".

Figure 3–1 ID page and Info page

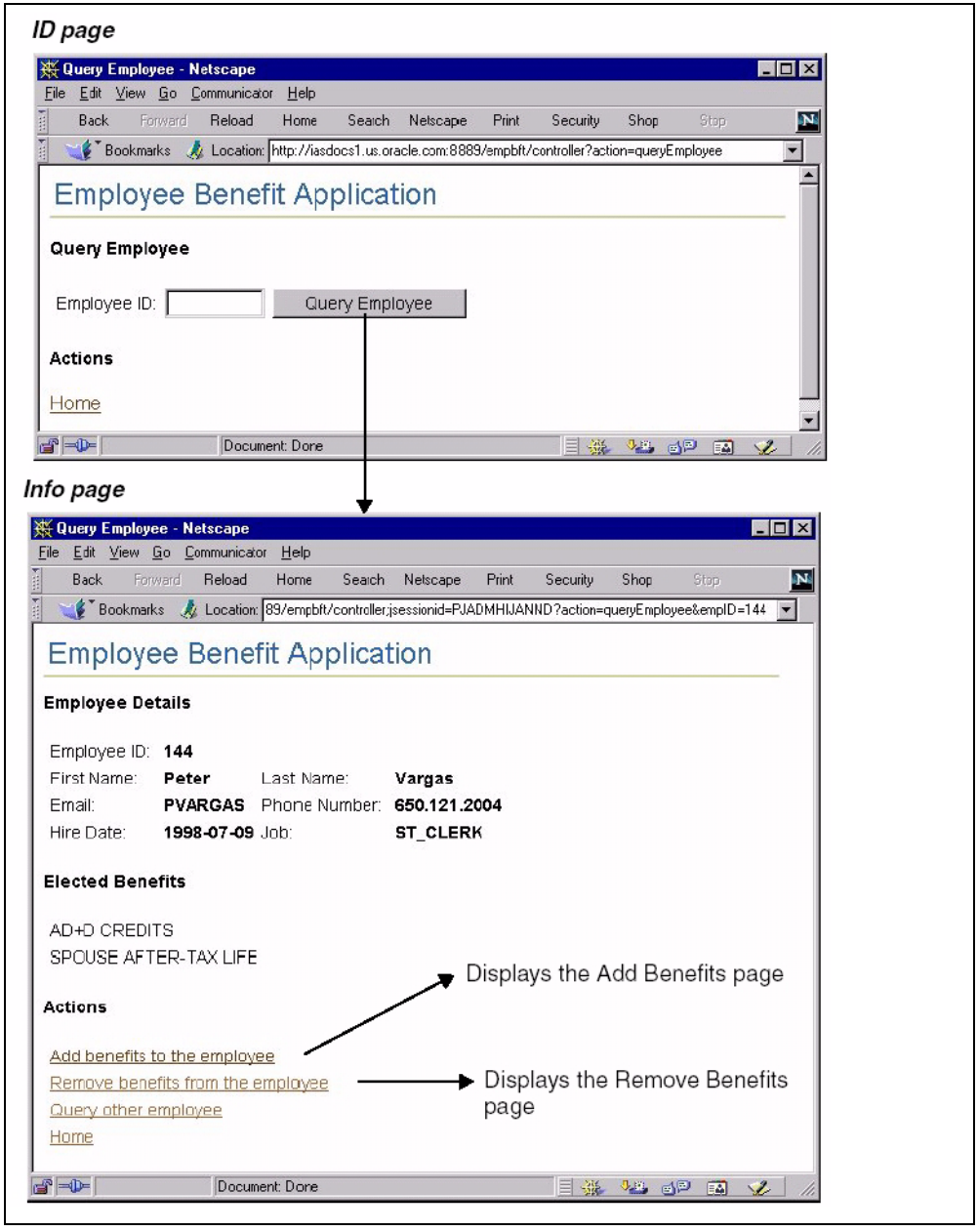


Figure 3–2 Add Benefits Page, Remove Benefits Page, and Success Page

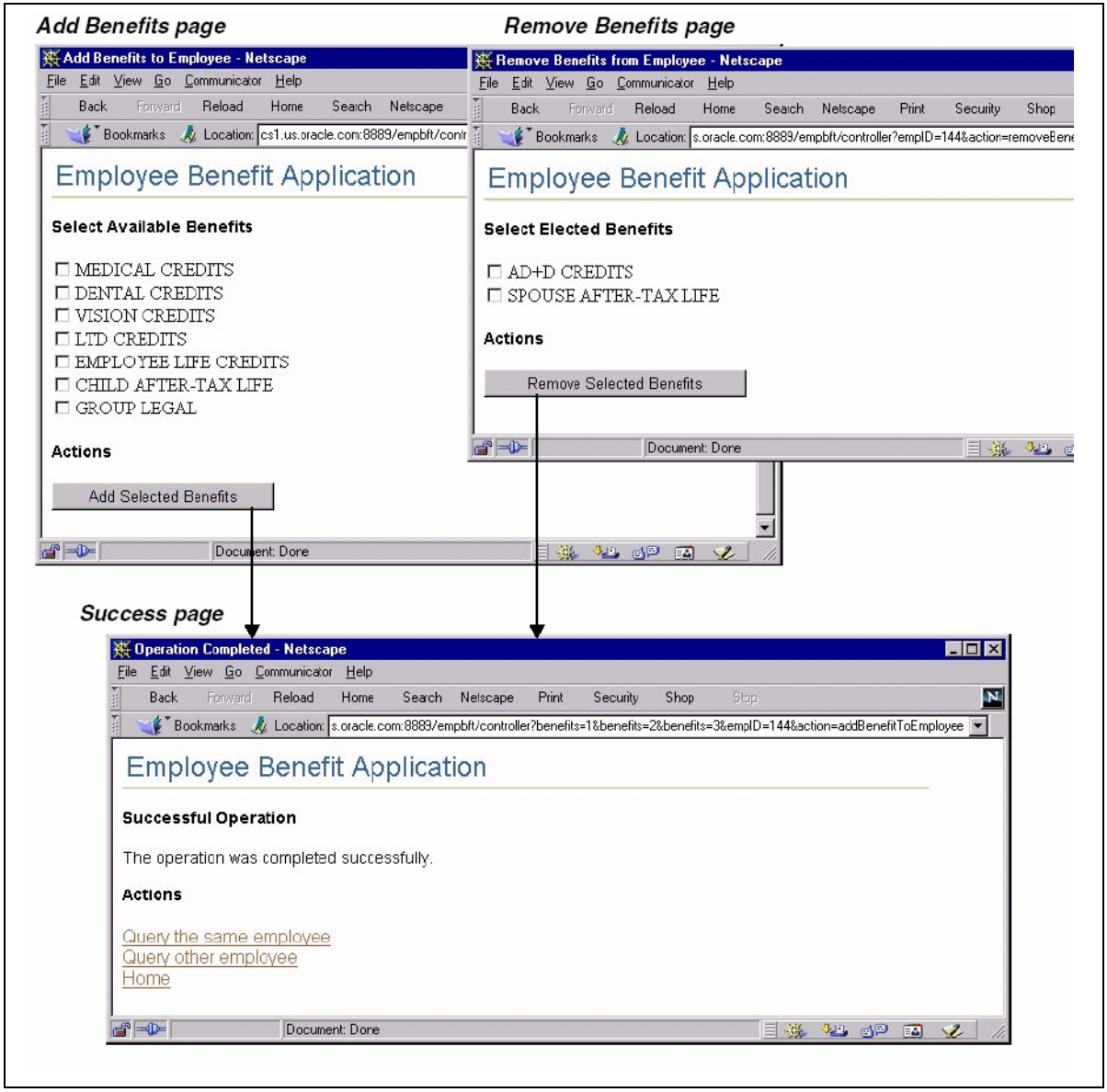
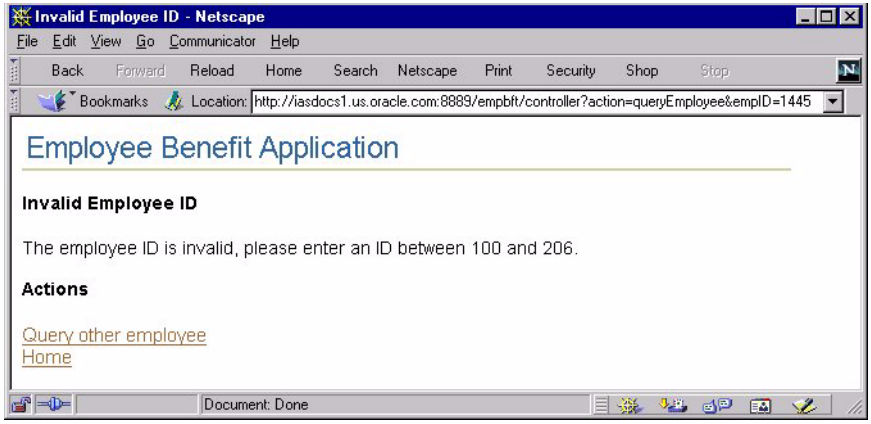


Figure 3–3 Error page



4

Implementing Business Logic

Recall that the first 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 first sample application consists of listing the employee information, adding benefits, and removing benefits (see Section 3.1, "Requirements for the First Sample Application") for a specific employee.

The database schema for the application, which you might find useful to review, is shown in Section 1.3, "Database Schema".

4.1 Objects Needed by the First Sample 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 first sample application uses EJBs because it might offer more functions to users in the future. The EJB container provides services that the application might need.

The first sample application needs the following EJBs:

- 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
*/
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();
    }
  }
}
```

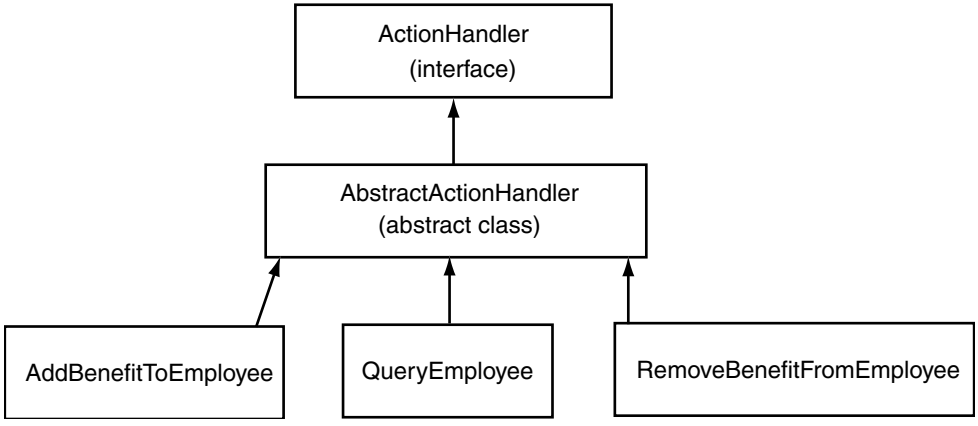
4.4 Action Handlers

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 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

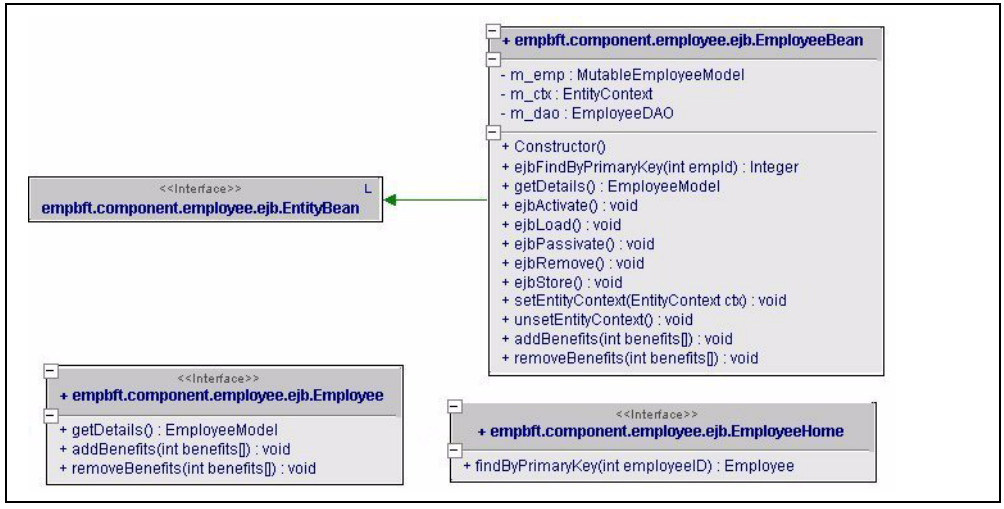
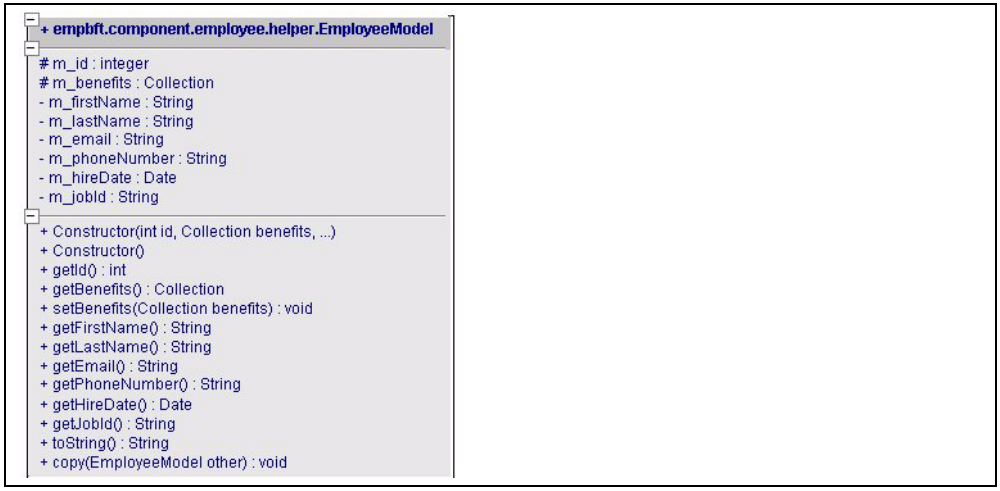


Figure 4–3 EmployeeModel Class



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:

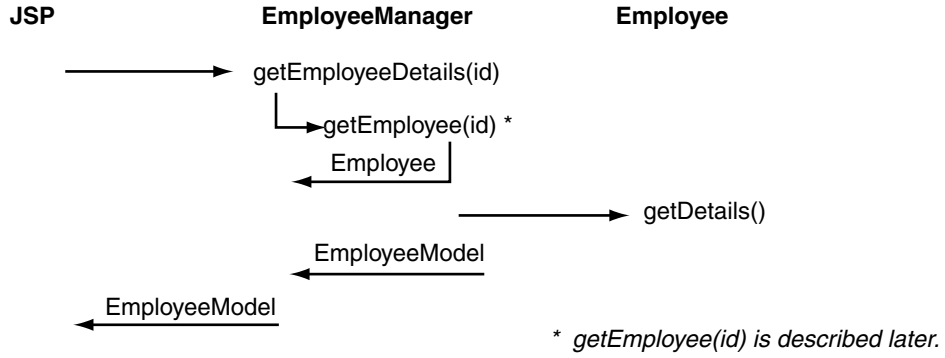
```
package empbft.component.employee.ejb;
import java.rmi.RemoteException;
import javax.ejb.EJBObject;
import empbft.component.employee.helper.*;

public interface Employee extends EJBObject
{
    public void addBenefits(int benefits[]) throws RemoteException;
    public void removeBenefits(int benefits[]) throws RemoteException;
    public EmployeeModel getDetails() 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 the `getEmployeeDetails` method in EmployeeManager, which in turn calls the `getEmployee` method (see Figure 4–4). The `getEmployee` method returns an Employee object, and the EmployeeManager invokes the `getDetails` method on this Employee object. The `getDetails` method returns an EmployeeModel object to the EmployeeManager, which returns it to the JSP. See Section 6.2, "Query Employee Operation" for 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.

See Chapter 10, "Updating EJBs to Use EJB 2.0 Features" for examples of EJBs that use container-managed persistence.

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

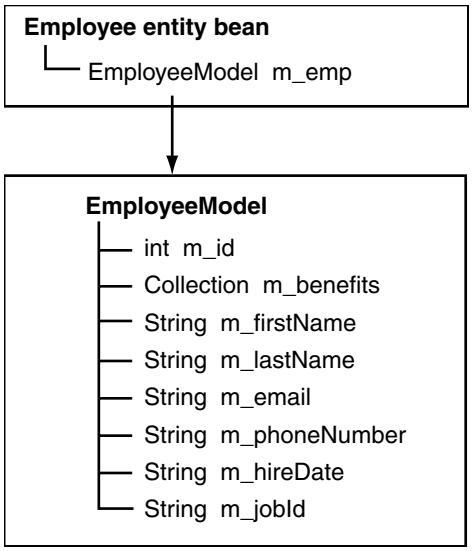
The implementation of the Employee bean uses a variable of type EmployeeModel, which contains all the employee details such as first name, last name, job ID, and so on. The following code snippet from EmployeeBean shows `m_emp` as a class variable:

```
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 (DAOs) 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 2–2.

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/blueprints/patterns/DAO.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 can be found in the EmployeeDAOImpl class. 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());
   }
```

```
}
```

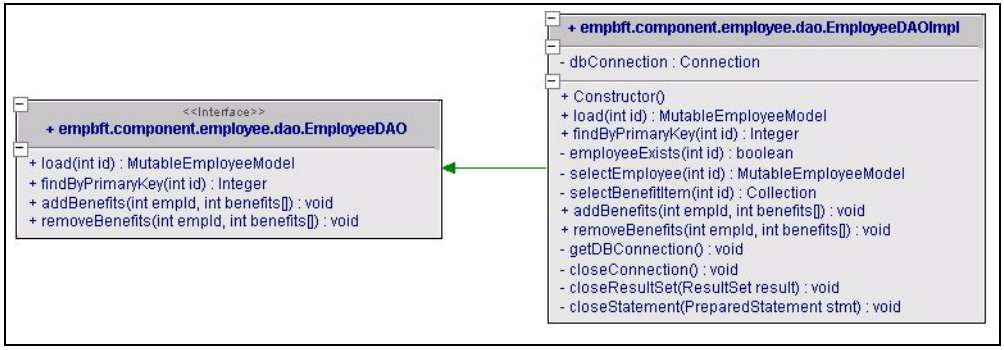
```
// also from EmployeeBean.java
public void ejbLoad()
{
   try {
      if (m_dao == null)
         m_dao = new EmployeeDAOImpl();
      Integer id = (Integer) m_ctx.getPrimaryKey();
      this.m_emp = m_dao.load(id.intValue());
   } catch (Exception e) {
      throw new EJBException("\nException in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



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.

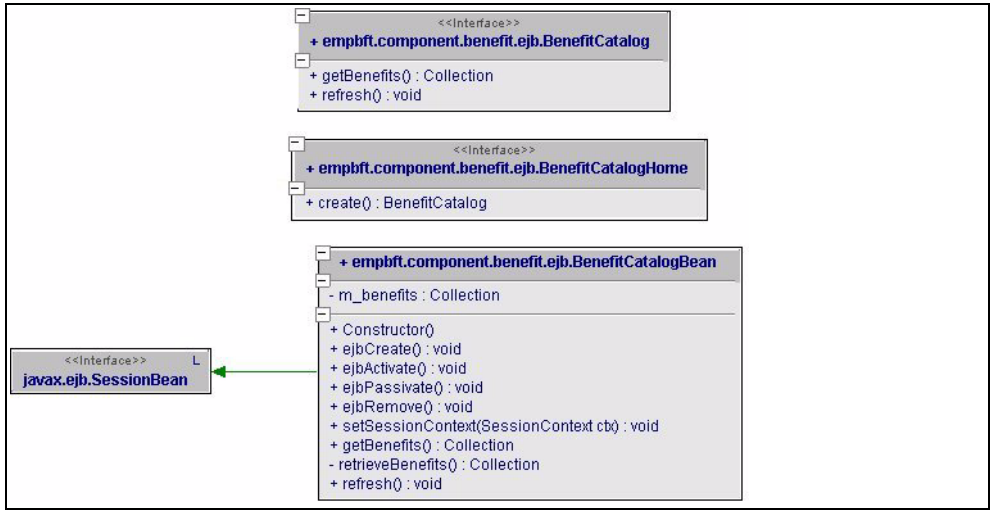
The application could have saved the benefit information to entity bean objects, but it uses a session bean instead. The reason for this is that the master benefit information does not change within the application. It is more efficient for a session bean to retrieve the data only once when the EJB container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



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;
}
```

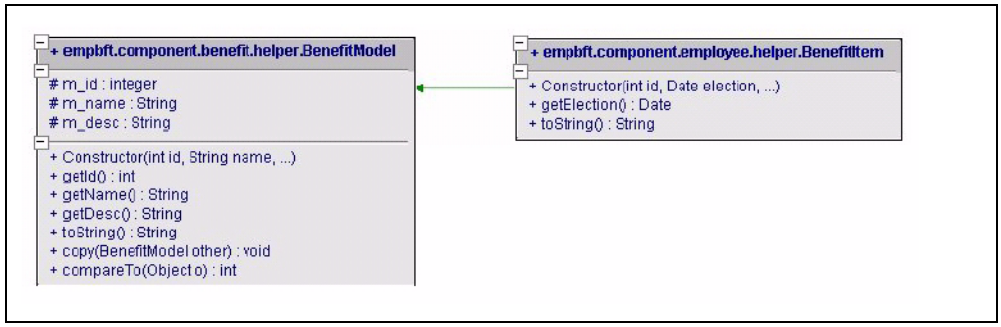
The `getBenefits` method returns a Collection of BenefitModels. This is the master list of all benefits. This method is called by the EmployeeManager bean (by the `getUnelectedBenefitItems` method) when the application needs to display a user's unelected benefits. It compares a user's elected benefits 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



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 not 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`:

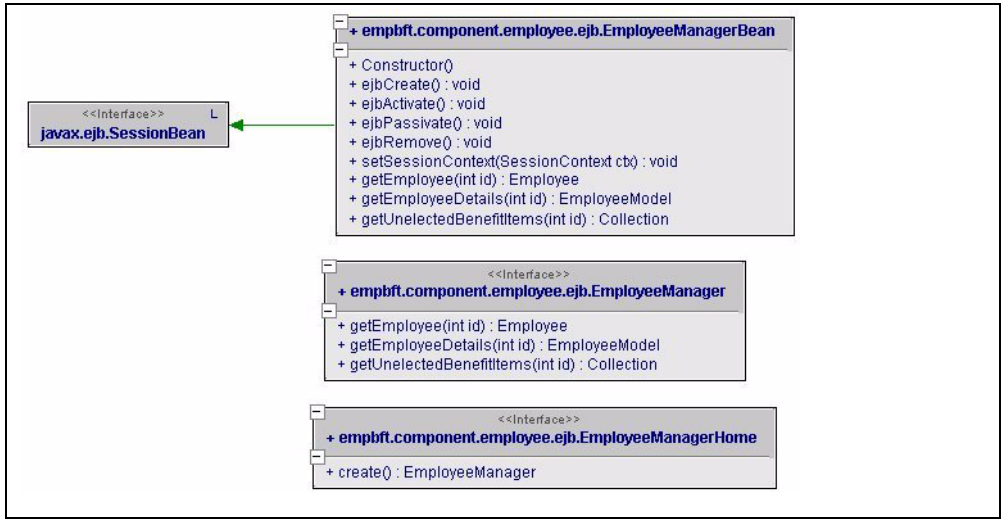
```
<%
  int empId = Integer.parseInt(request.getParameter(
                              SessionHelper.EMP_ID_PARAMETER));
  EmployeeManager mgr = SessionHelper.getEmployeeManager(request);
  Collection unelected = mgr.getUnelectedBenefitItems(empId);
  ...
%>
```

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



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
<%
```

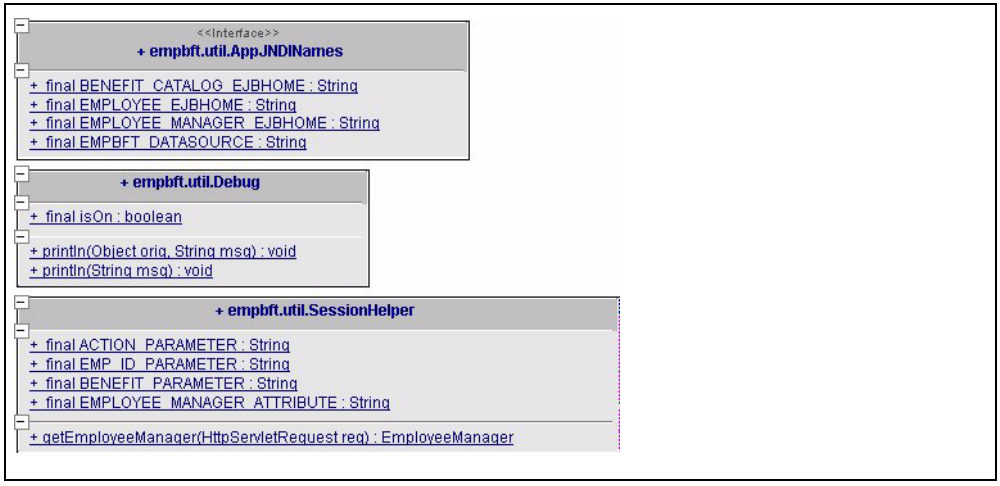
```
  int empId = Integer.parseInt(request.getParameter(
                          SessionHelper.EMP_ID_PARAMETER));
  EmployeeManager mgr = SessionHelper.getEmployeeManager(request);
  Collection elected = mgr.getEmployeeDetails(empId).getBenefits();
  ...
%>
...
<h4>Select Elected Benefits</h4>
<%
   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



5

Creating Presentation Pages

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

Contents of this chapter:

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, OracleAS 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 the *Oracle Application Server Containers for J2EE Servlet Developer's Guide* for details on servlets.

5.2.1 Automatic Compilation of Servlets

One advantage to updating servlets is that OracleAS has an auto-compile feature for servlets. You can place the uncompiled .java files for the servlets in the `$J2EE_HOME/default-web-apps/WEB-INF/classes` directory, and OracleAS will compile the files for you. To enable the auto-compile feature, set the `development` attribute of the `orion-web-app` tag to "`true`". This tag is found in `$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 the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* 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 the *Oracle Application Server Containers for J2EE 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 sample 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

Tracing Flows 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:

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:

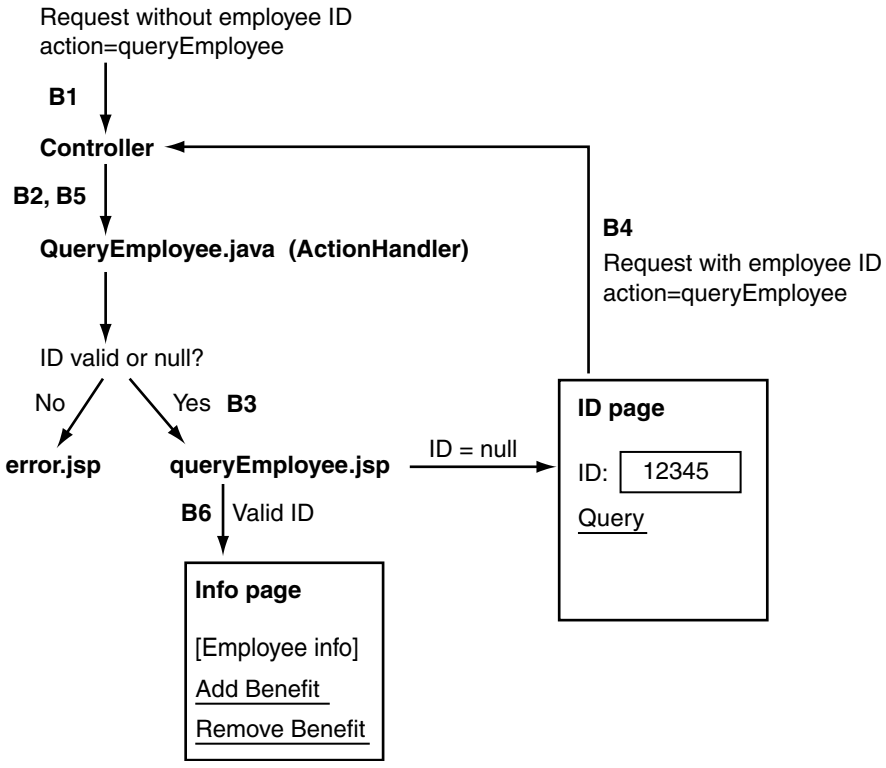
```
http://<host>/empbft/controller?action=queryEmployee
```

The user then sees the ID page (Figure 3–1).

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 8.5.1, "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 3–1).

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

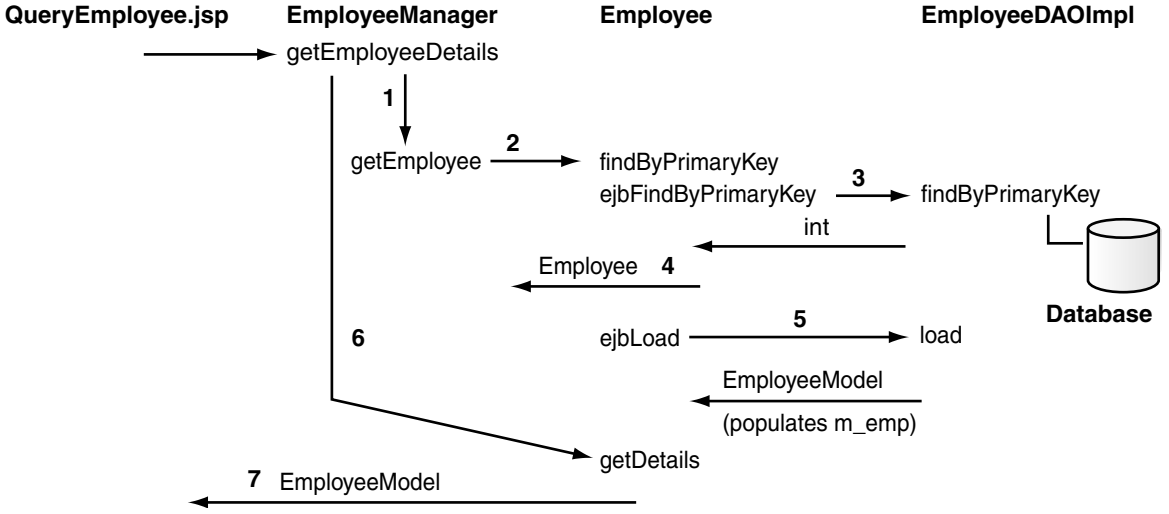
To get employee details, `queryEmployee.jsp` invokes the `getEmployeeDetails(employeeId)` method in EmployeeManager. The method returns an EmployeeModel object, which contains the data. The JSP then retrieves values from the EmployeeModel object to display the employee data.

```
// 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 `findByPrimaryKey`, 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.

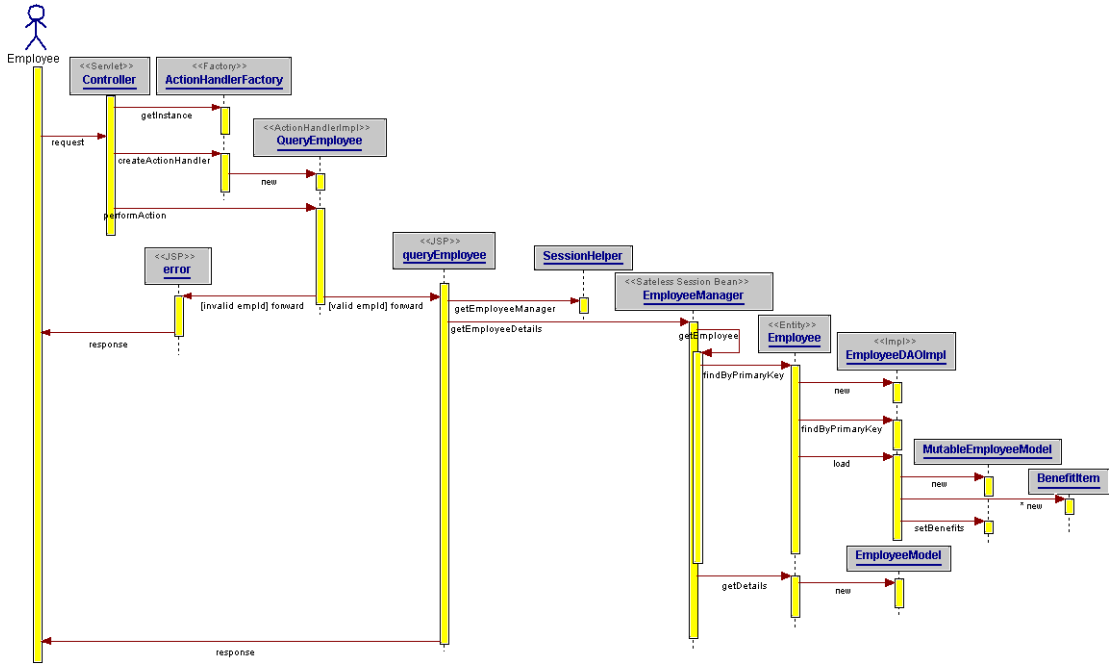
6.2.4 Getting Benefit Data

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 3–1). 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 3–1). The URL behind this link looks like:

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

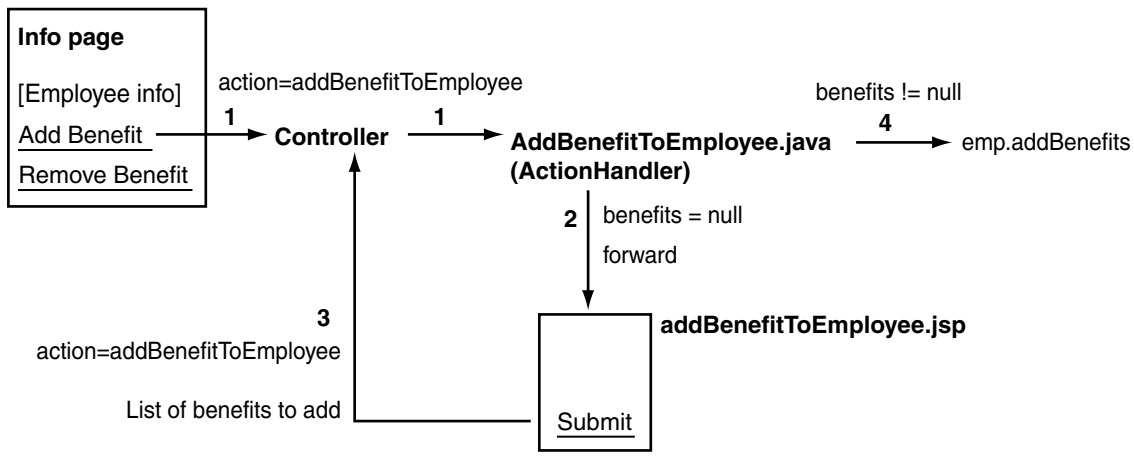
See the following sections for details on the add and remove operations.

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.

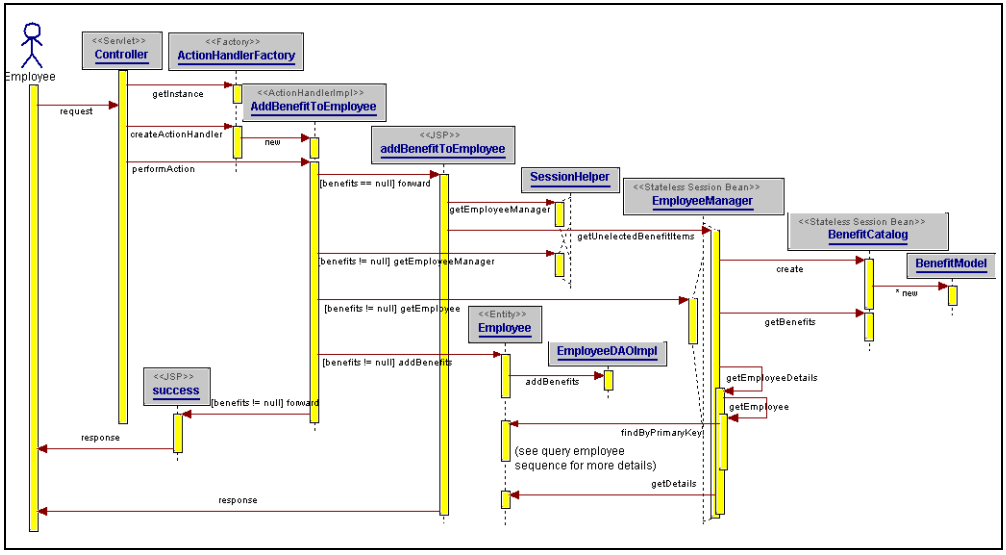
```
// from EmployeeBean.java
public void addBenefits(int benefits[])
{
    try{
      if (m_dao == null) m_dao = new EmployeeDAOImpl();
      m_dao.addBenefits(m_emp.getId(), benefits);
      ejbLoad();
    } catch (Exception e) {
      throw new EJBException ("\nData access exception in adding benefits.\n"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

```
            + 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

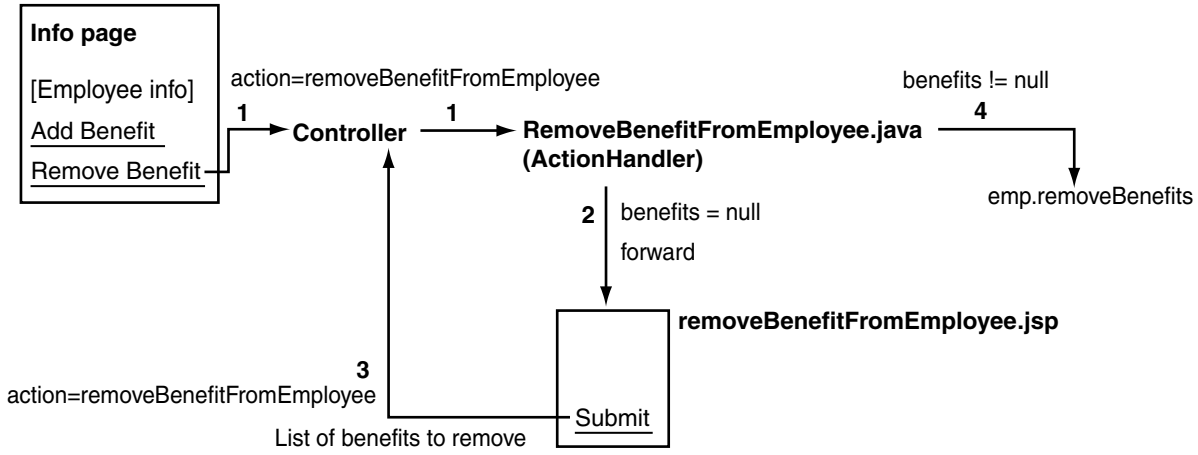


6.5 Removing Benefit Operation

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



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.

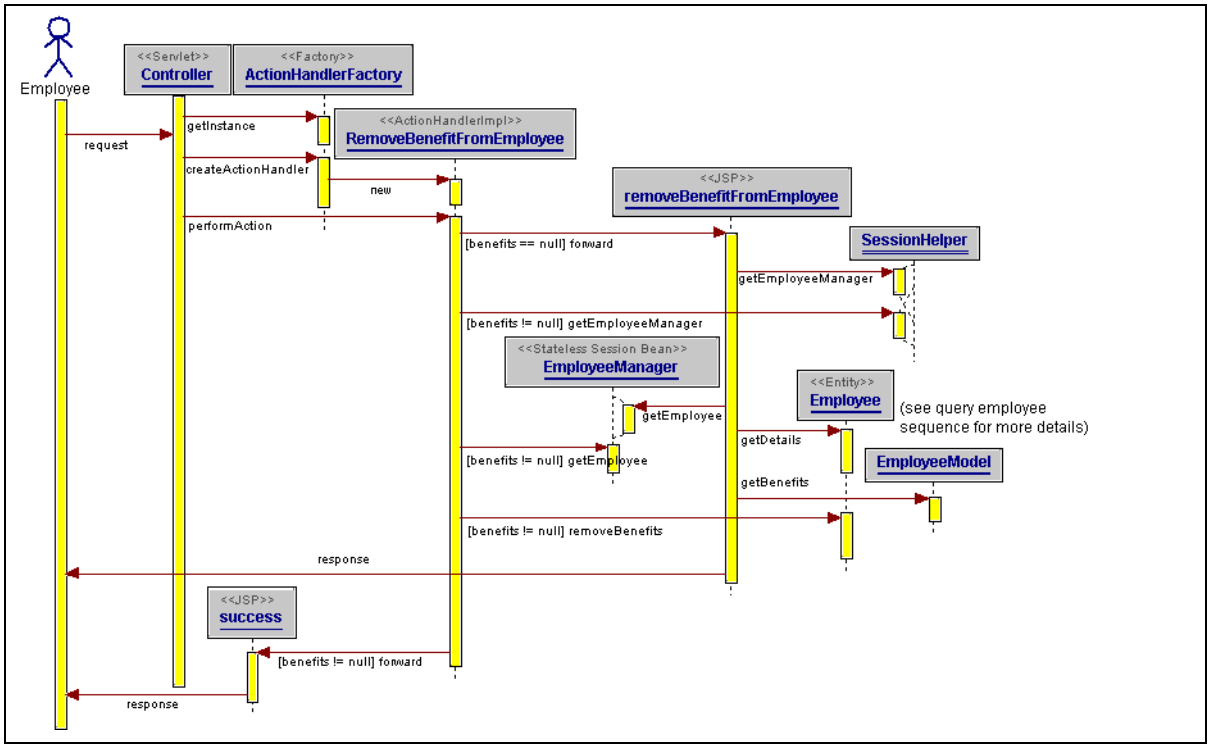
```
// from EmployeeBean.java
public void removeBenefits(int benefits[])
{
```

```
   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

Configuring OracleAS Web Cache for the Application

You can use OracleAS Web Cache 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 OracleAS Web Cache Manager tool.

This guide does not cover how OracleAS Web Cache works. To learn about OracleAS Web Cache, see the *Oracle Application Server Web Cache Administrator's Guide*.

7.1 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 OracleAS 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 OracleAS 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 sample application caches static and dynamic pages.

7.2 Analyzing the Application

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

The ID page (Figure 3–1), 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.

7.2.1 Specifying the Pages to Cache

Figure 7–1 shows the OracleAS Web Cache Manager with the Cacheability, Personalization, and Compression Rules page displayed. The first three rows apply to the sample application.

Caching the ID Page and the Employee Information Page

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.*
```

Caching the Style Sheet

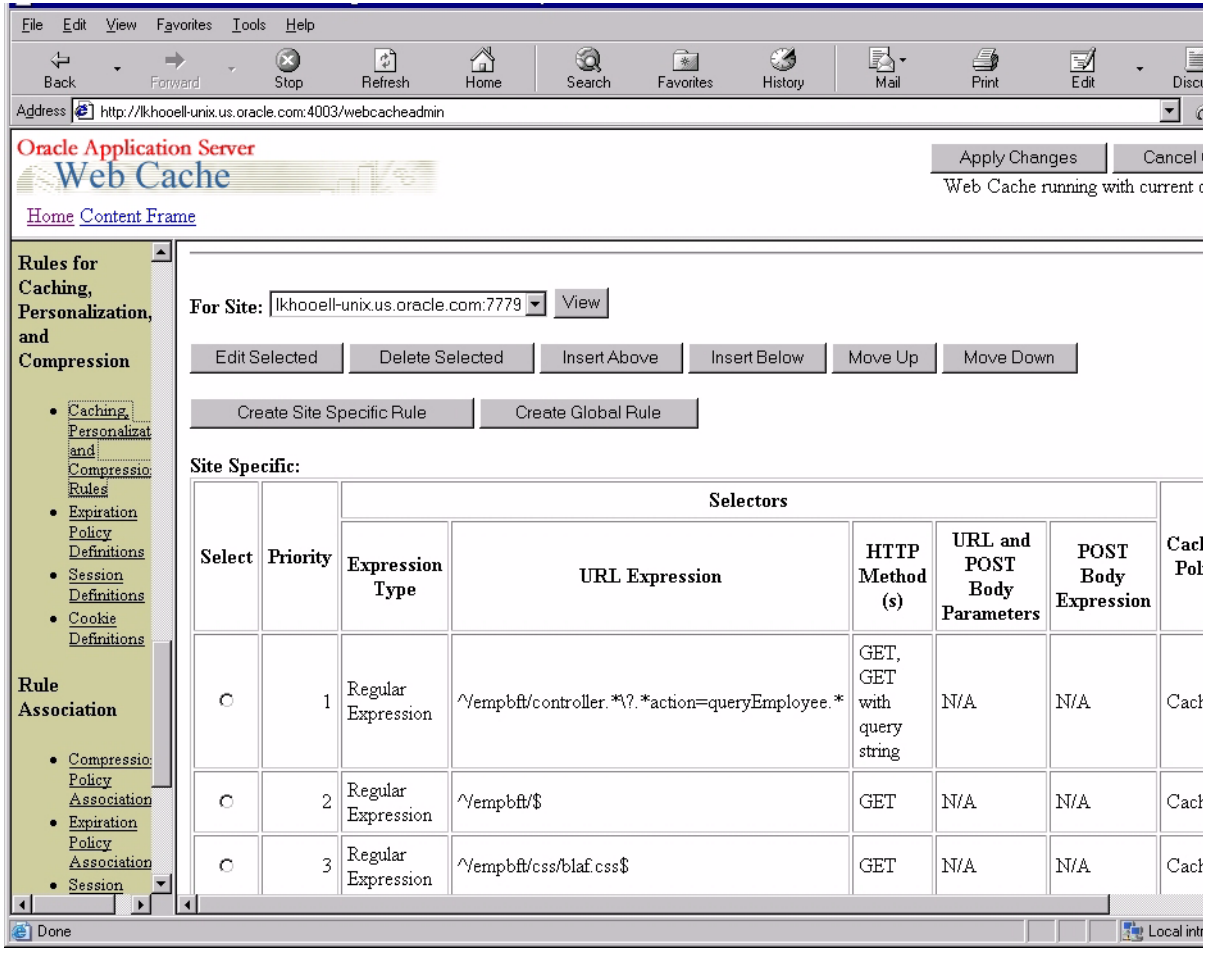
To cache the style sheet, specify its URL in the OracleAS Web Cache Manager. The ^ and $ are special characters used in regular expressions to indicate the beginning and the end of a line. This ensures that the pattern matches exactly.

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

Caching the First Page

The second rule, `^/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 7–1 Caching, Personalization, and Compression Page in OracleAS Web Cache Manager



7.2.2 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 the OracleAS Web Cache invalidation port.

The 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 OracleAS 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 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 OracleAS 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 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 OracleAS 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
```

OracleAS Web Cache listens on a specific port. The procedure calls `utl_tcp.open_connection` to open a connection to OracleAS 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 OracleAS Web Cache.

7.2.3 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;
```

8

Supporting Wireless Clients

The wireless feature in OracleAS 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 the sample application to support wireless clients.

Contents of this chapter:

8.1 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 8.2, "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 8.3.3, "Separating Presentation Data into Separate Files".

8.2 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 OracleAS transforms the XML to the specific markup language that the wireless client can process.

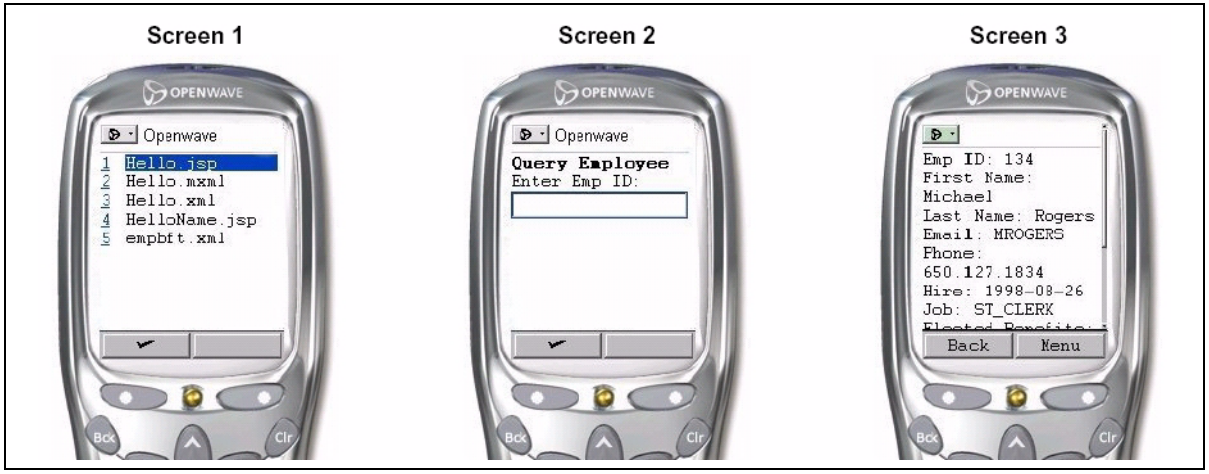
Like HTML, applications can generate XML from JSP files or static files. In the sample application, the presentation data comes from JSP files because it contains dynamic data. See Chapter 5, "Creating Presentation Pages".

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 *Oracle Application Server Wireless Developer's Guide*.

8.2.1 Screens for the Wireless Application

Figure 8–1 to Figure 8–3 show the sample application on an OpenWave simulator. The application on a wireless client looks similar to the application on a desktop browser.

Figure 8–1 Screens for the Wireless Application (1 of 3)



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

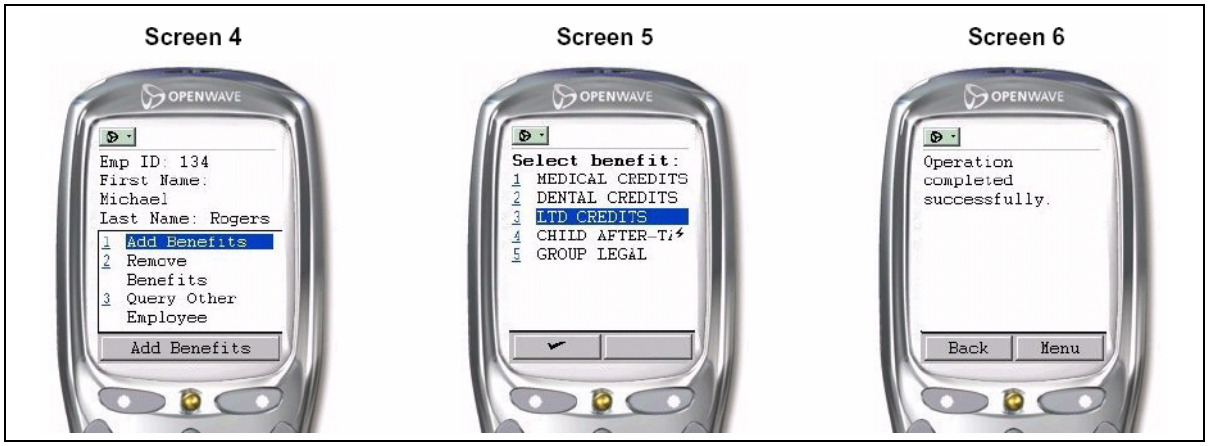
```
$OMSDK_HOME/oc4j_omsdk/omsdk/j2ee/applications/pmsdk/apps/
```

$OMSDK_HOME is the home directory for OracleAS 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 8–2 Screens for the Wireless Application (2 of 3)

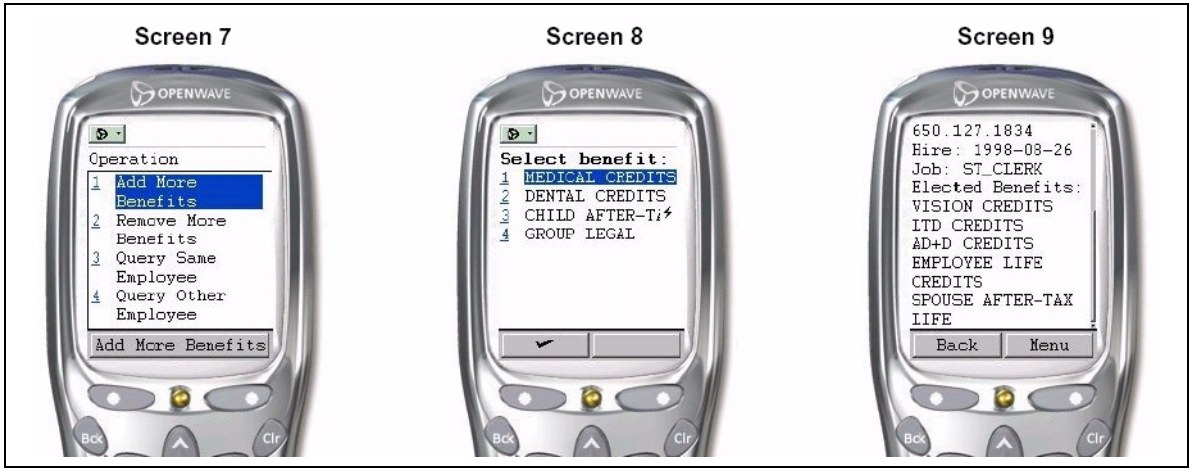


Screen 4 shows the menu, which offers selections such as add benefits, remove benefits, and query other employee.

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 8.2.2, "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 8–3 Screens for the Wireless Application (3 of 3)



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.

8.2.2 Differences Between the Wireless and the Browser Application

In the browser version of the 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 necessary, and thus not available, for the browser version of the application.

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).

8.3 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.

8.3.1 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);
%>
```

8.3.2 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)
%>
```

8.3.3 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>
```

8.4 Header Information in JSP Files for Wireless Clients

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

8.4.1 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"?>
```

8.4.2 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.

8.5 Operation Details

To OracleAS, 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 OracleAS 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 OracleAS Wireless.

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

OracleAS Wireless then invokes a transformer, which takes the XML and transforms it to a markup language appropriate for the wireless client. OracleAS 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 *Oracle Application Server Wireless Developer's Guide* and the *Oracle Application Server Wireless Administrator's Guide* for further details.

8.5.1 Query Operation

Figure 8–4 shows the flow of the query operation with wireless and browser clients. This figure is a more complex form of Figure 6–1.

Figure 8–4 Query Operation

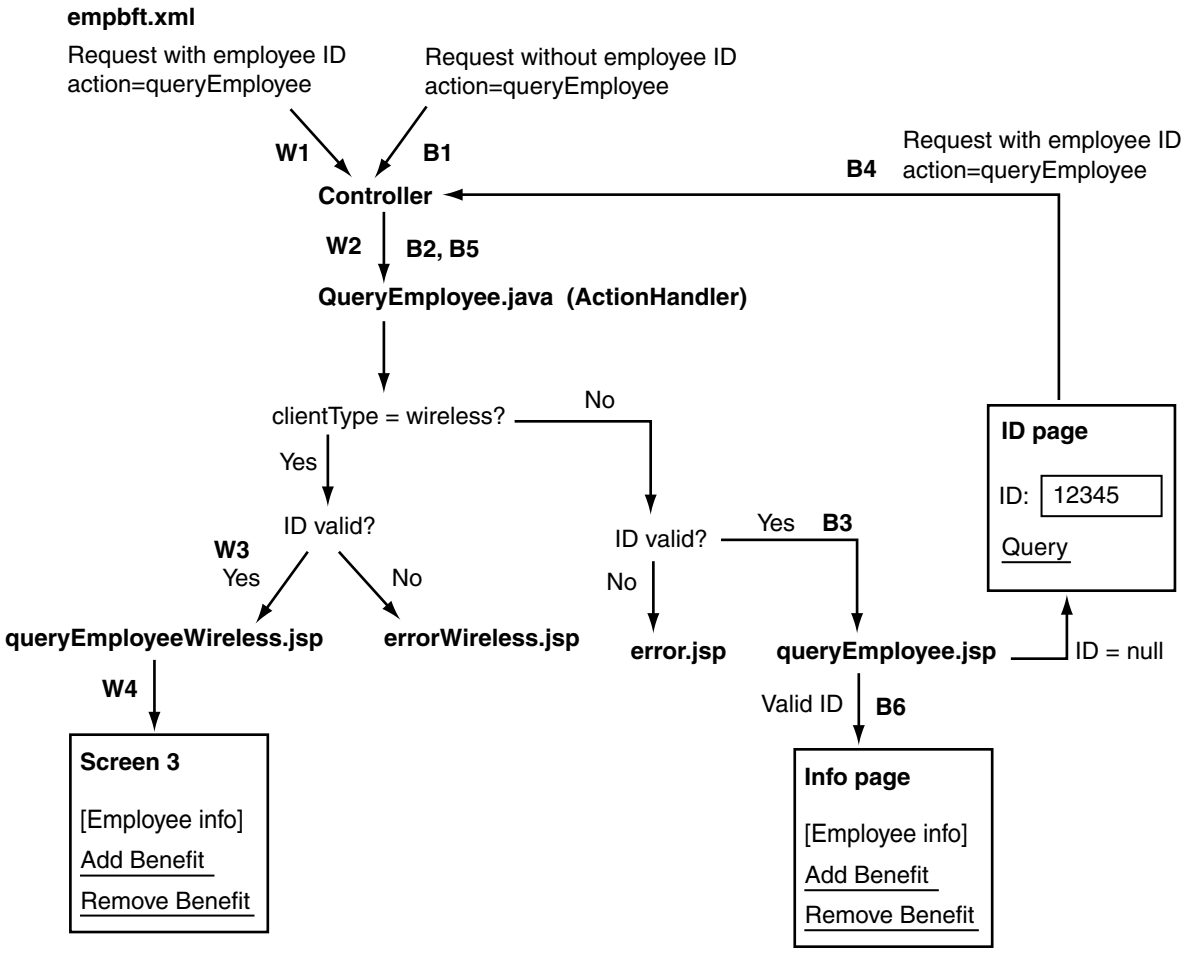


Figure 8–4 contains two sequences of events. One sequence is for requests that come from browsers; steps in this sequence are noted in the figure with a "B". The other sequence is for requests that come from wireless clients; steps in this sequence are noted with a "W".

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).

8.5.2 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 8–1, Screen 4). The user presses the Menu button to access the popup menu.

8.5.3 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 8.2.2, "Differences Between the Wireless and the Browser Application" describes this feature in detail.

8.6 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.

8.6.1 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 8–1, Screen 1) that lists all the applications in a certain directory (see Section 8.2.1, "Screens for the Wireless Application").

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

8.6.2 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 the application can be accessed publicly (that is, it is not behind a firewall). Requests from wireless clients go through gateways, which can communicate only with machines that are accessible publicly.

Your application should then appear on the list of applications when you enter the URL and follow the steps listed in Section 8.6.1, "Using a Simulator".

9

Running in a Portal Framework

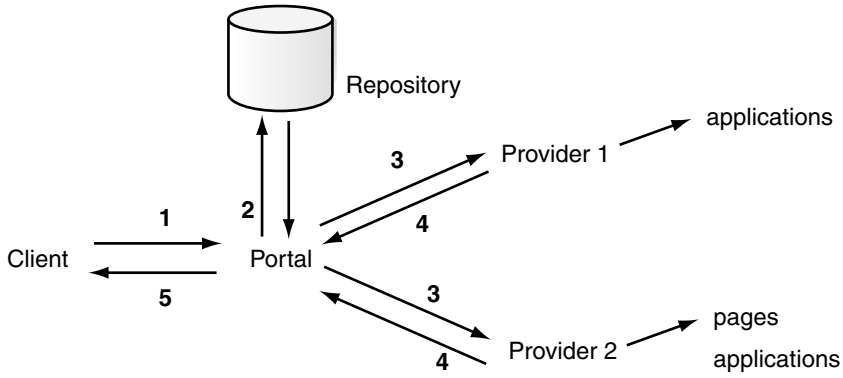
To make the sample application run within a portal framework, you have to make some changes to the application. The changes that you have to make are in the controller and the action handler objects. You also have to edit the links in the JSP files to make them work. The model layer (that is, the Employee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

Topics in this chapter:

9.1 How Portal Processes Requests

Figure 9–1 shows how portal handles requests. This is important in understanding why you have to use APIs in the Oracle Application Server Portal Developer Kit to code your links and parameters.

Figure 9–1 How Portal Processes Requests



1. A client sends a request to OracleAS 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 9–6 show the pages of the application in a portal. You can compare these portal pages with the non-portal pages in Figure 3–1 and Figure 3–2.

Figure 9–2 ID Page in a Portal

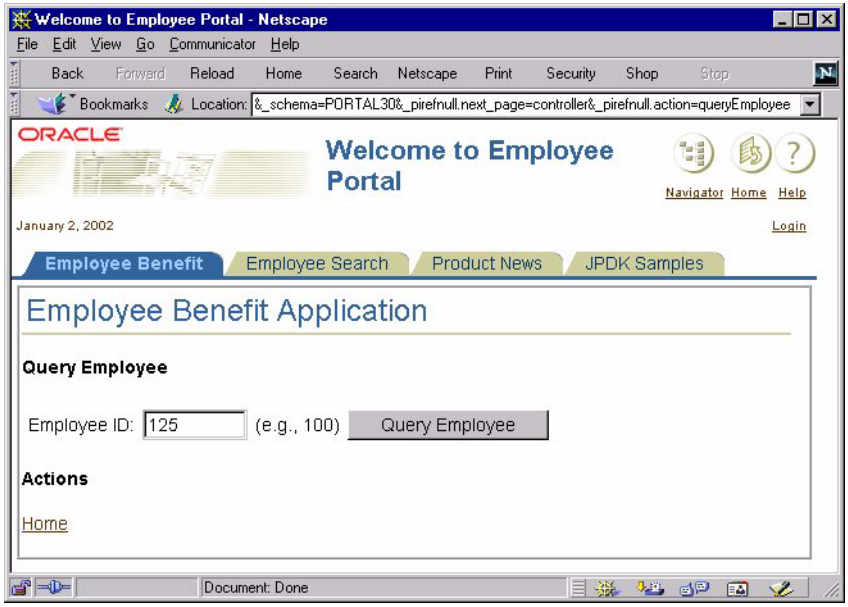


Figure 9–3 Info Page in a Portal

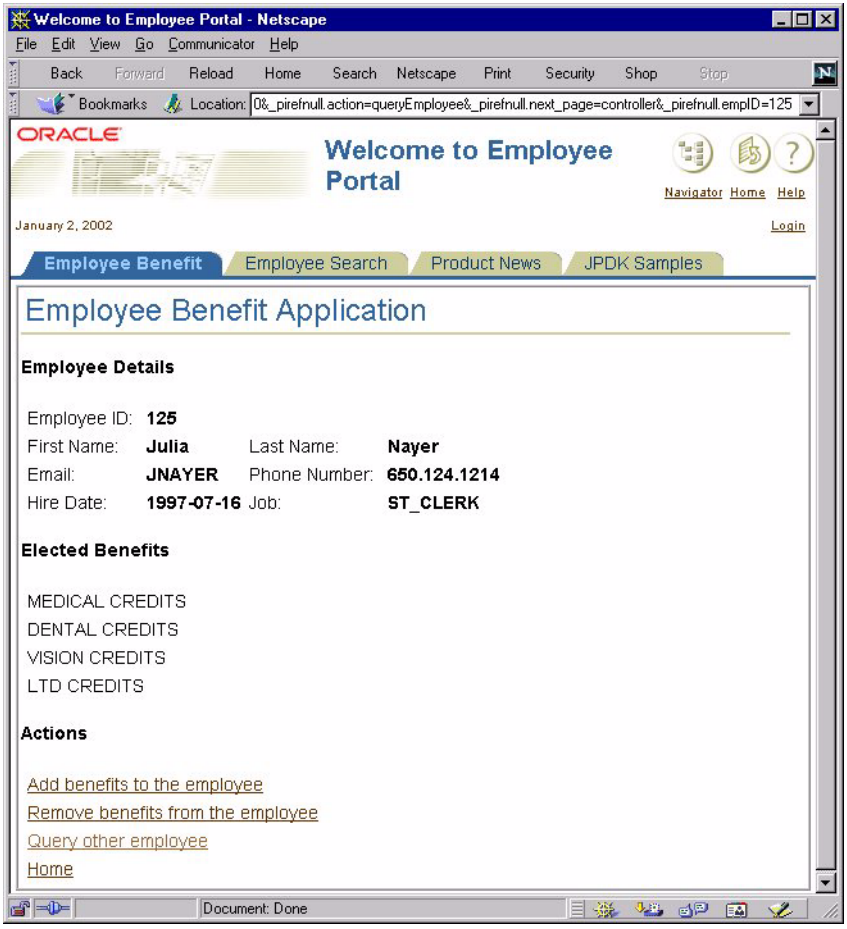


Figure 9–4 Add Benefits Page in a Portal

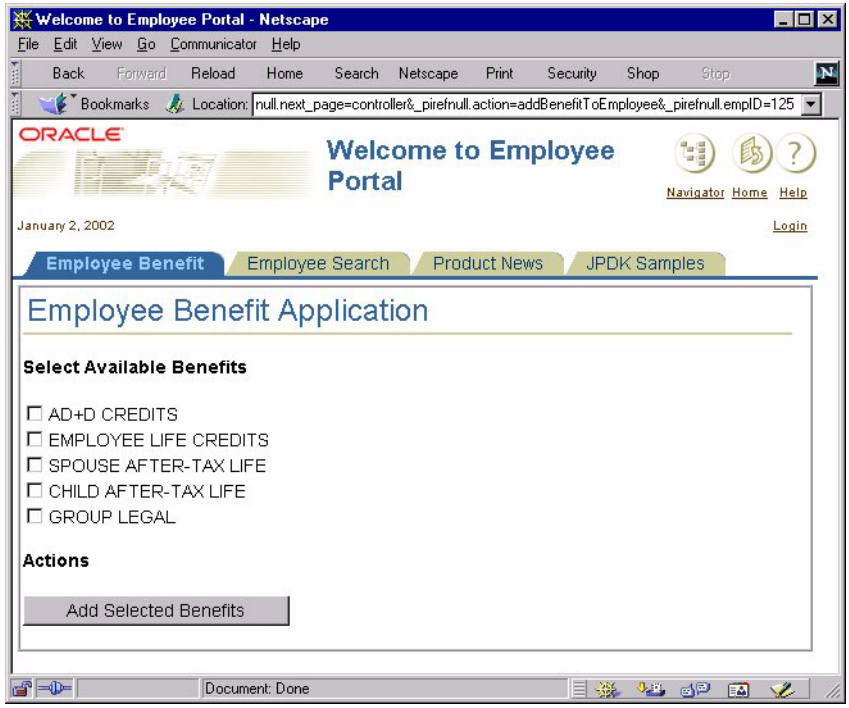


Figure 9–5 Remove Benefits Page in a Portal

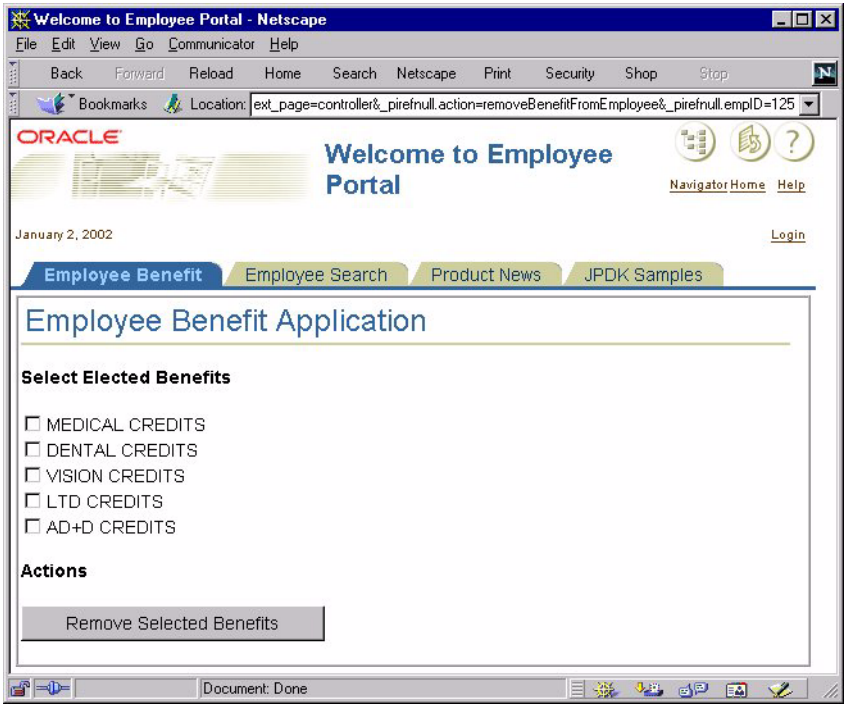
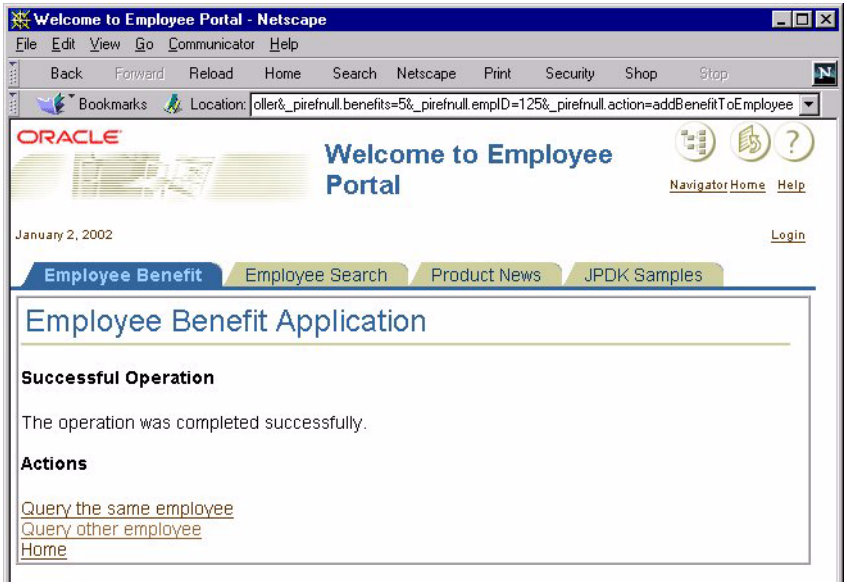


Figure 9–6 Success Page in a Portal



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.

Figure 9–7 shows a sample portal page that contains the 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 9–7 A Portal Page Containing the Sample 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 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, 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 application sends the request to the controller object. This chapter explains why you have to set up the link this way.

After OracleAS 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 OracleAS 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.

Part III

The Second Sample Application

This part of the guide describes the second sample application. It contains the following chapters:

- Chapter 10, "Updating EJBs to Use EJB 2.0 Features"
- Chapter 11, "Enabling Web Services in the Application"

10

Updating EJBs to Use EJB 2.0 Features

This chapter describes the implementation details of the second sample application. The first part of the chapter describes the business methods in the application. The second part describes how the EJBs in the application map to database tables.

Contents of this chapter:

10.1 Overview of the Second Sample Application

The second sample application provides business operations that clients can invoke through Web Services. The third sample application (described in Chapter 11, "Enabling Web Services in the Application") is an example of such a client.

The business operations involve accessing and updating data in the HR schema, which is the same schema used in the first sample application.

The application uses EJBs to implement the business operations. The EJBs use features such as container-managed persistence, EJB query language, local interfaces, and container-managed relationships.

Note that the second sample application does not contain any JSPs or servlets because it does not display any pages. The application simply contains EJBs and some supporting Java classes. Its operations are accessed by clients through Web Services, and it is the client applications that display the results.

10.1.1 Business Operations in the Second Sample Application

The second sample application implements these business operations:

- List all benefits
- Add new benefits
- Count the number of employees
- List the benefits an employee has
- Count the number of employees enrolled in a specified benefit

To implement the operations, the application uses entity beans and a session bean:

Table 10–1 EJBs in the Second Sample Application

EJB	Description
Employee (entity bean)	The Employee entity bean contains fields for employee information such as employee ID, first name, last name, address, and benefits the employee has selected.
Benefit (entity bean)	The Benefit entity bean contains fields for benefit information such as benefit ID, benefit name, benefit description, and employees who have that benefit.
EmployeeBenefitManager (session bean)	The EmployeeBenefitManager EJB is a stateless session bean that provides the interface for the application's clients. Clients call methods in the EmployeeBenefitManager to access the application's business operations. These methods, in turn, invoke methods in the Employee and Benefit entity beans. Clients do not call the Employee and Benefit beans directly.

Later sections in this chapter describe each business operation in detail.

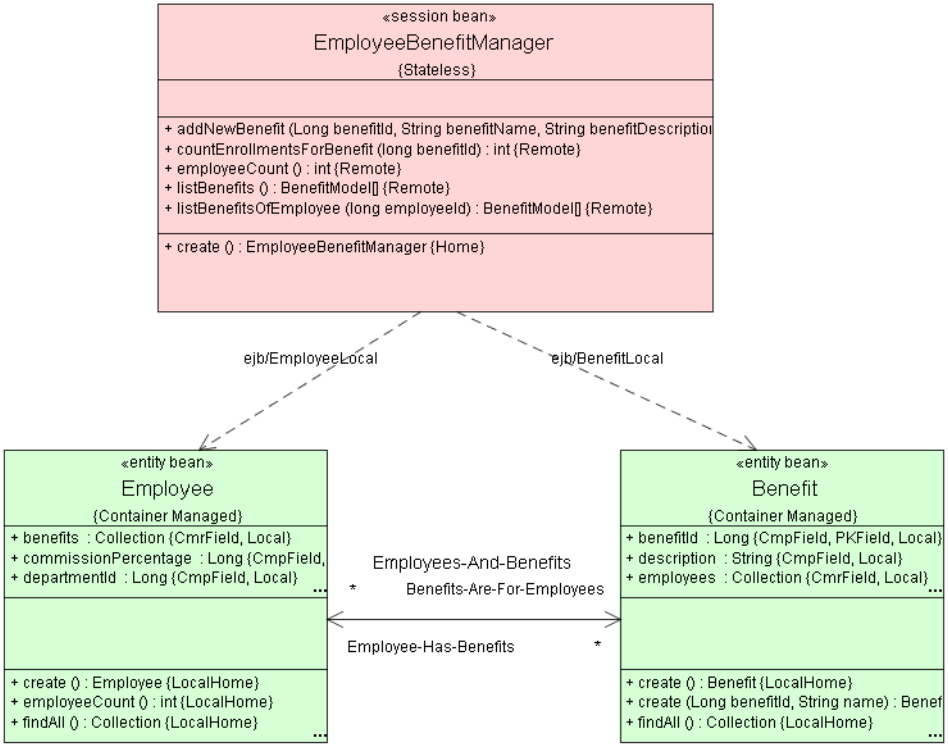
Entity beans provide a flexible model: they can create new tables and columns and populate them, or they can work with existing tables. In the sample application, they work with existing tables.

Figure 10–1 shows how the EJBs in the application work together. It shows:

- business methods in the EmployeeBenefitManager session bean

- some of the fields and methods in the Employee and Benefit entity beans
- an employee can be associated with zero or more benefits, and a benefit can be associated with zero or more employees

Figure 10–1 UML Diagram for the EJBs



10.1.2 Design of the Second Application

The second sample application follows the "session facade" design pattern. This means that the methods in the entity beans are hidden from clients. Instead, clients call methods in the EmployeeBenefitManager session bean to invoke business operations. Methods in the session bean invoke other methods in the entity beans. Clients do not even know about the existence of the entity beans.

10.1.3 EJB 2.0 Features Used by the Entity Beans

The entity beans use features from the EJB 2.0 specification. In particular, they use these features:

- container-managed persistence (CMP) instead of bean-managed persistence (BMP)
- container-managed relationships between employees and their benefit items
- local interfaces instead of remote interfaces
- EJB Query Language (QL)

Note that these features do not apply to the EmployeeBenefitManager session bean. This session bean uses a remote interface (not local interface) so that remote clients can invoke it. Remote clients are clients that run in a different JVM or in a different application. If the session bean used a local interface, then only clients running in the same application can invoke it.

Persistent Fields in the Entity Beans

The entity beans contain fields for data that are stored in database tables. Examples of fields in the Employee entity bean are Employee ID, First Name, Last Name, and Hire Date. Examples of fields in the Benefit entity bean are Benefit ID, Benefit Name, and Description.

To enable the EmployeeBenefitManager session bean to access the fields in the entity beans, you do these tasks for each field:

- Declare accessor methods (get and set methods) for each field in the local interface (`EmployeeLocal.java` and `BenefitLocal.java`).

  For example, for the First Name field, you would have the `getFirstName` and `setFirstName` methods. See Section 10.7.2, "Persistent Fields in the Local Interface" for details.

- Create abstract methods for the accessor methods in the bean implementation class (`EmployeeBean.java` and `BenefitBean.java`).

  The container takes the abstract methods and implements the code to perform the get or set operation. The container connects to the database to get or set the values.

- Map the fields to table columns in the `orion-ejb-jar.xml` file. This is required because the field names do not match the names of the table columns in the database. See Section 10.7.3, "Persistent Fields in the orion-ejb-jar.xml File" for details.

10.2 Details of employeeCount Method

To get the number of employees, a client calls the `employeeCount` method in the EmployeeBenefitManager session bean. This method calls the corresponding method in the Employee entity bean. The client does not invoke the method in the Employee entity bean directly.

```
// EmployeeBenefitManagerBean.java
public int employeeCount()
{
   try
   {
      return getEmployeeLocalHome().employeeCount();
   }
   catch(Exception e)
   {
      e.printStackTrace();
      throw new EJBException(e);
   }
}

// The following method gets the local home object for the Employee bean.
private EmployeeLocalHome getEmployeeLocalHome() throws NamingException
{
   final InitialContext context = new InitialContext();
   return (EmployeeLocalHome)context.lookup("java:comp/env/ejb/EmployeeLocal");
}
```

In the EmployeeLocalHome interface, the `employeeCount` method executes the `ejbHomeEmployeeCount` method in the EmployeeBean class:

```
// EmployeeBean.java
public int ejbHomeEmployeeCount()
{
   try
   {
      return ejbSelectAllEmployees().size();
   }
   catch(FinderException fe)
   {
      return 0;
   }
}
```

```
public abstract Collection ejbSelectAllEmployees() throws FinderException;
```

The `ejbSelectAllEmployees` method uses EJB QL to get its results. The name of the method and its EJB QL statement are defined in the `ejb-jar.xml` file:

```
// ejb-jar.xml
<entity>
   <description>Entity Bean ( CMP )</description>
   <display-name>Employee</display-name>
   <ejb-name>Employee</ejb-name>
   ... lines omitted ...
   <abstract-schema-name>Employee</abstract-schema-name>
   ... lines omitted ...
   <query>
      <query-method>
         <method-name>ejbSelectAllEmployees</method-name>
         <method-params/>
      </query-method>
      <ejb-ql>select object(e) from Employee e</ejb-ql>
   </query>
   ... lines omitted ...
</entity>
```

The `<method-name>` element specifies the name of the method, and the `<ejb-ql>` element specifies the EJB QL statement to execute when the method is invoked. The `<abstract-schema-name>` element specifies the name to use in EJB QL statements to identify entity beans.

In the EJB QL statement, the `Employee` reference matches the name specified in the `<abstract-schema-name>` element. The statement selects all Employee entity beans and returns a `Collection` of Employee beans to the `ejbHomeEmployeeCount` method. The method uses the `size` method to determine the number of elements in the `Collection`.

10.3 Details of listBenefits Method

To get a list of all benefits, a client calls the `listBenefits` method in the EmployeeBenefitManager session bean. This method calls the `findAll` method in the Benefit bean.

```
// EmployeeBenefitManagerBean.java
public BenefitModel[] listBenefits()
{
   int count = 0;
```

```
   BenefitModel[] returnBenefits;
   Collection allBenefits = null;
   BenefitLocal benefitLocal = null;
   BenefitModel benefit = null;

   try
   {
      allBenefits = getBenefitLocalHome().findAll();
      returnBenefits = new BenefitModel[allBenefits.size()];
      Iterator iter = allBenefits.iterator();
      while(iter.hasNext())
      {
         benefitLocal = (BenefitLocal)iter.next();
         benefit = new BenefitModel(benefitLocal.getBenefitId(),
                                    benefitLocal.getName(),
                                    benefitLocal.getDescription());
         returnBenefits[count++]=benefit;
      }
      return returnBenefits;
   }
   catch(Exception e)
   {
      e.printStackTrace();
      throw new EJBException(e);
   }
}
```

The `findAll` method is a special method. You declare the method in the BenefitLocalHome interface, but do not implement it in the BenefitBean class. The container implements it for you. You do not have to define a query statement for the method in the `ejb-jar.xml` file (it gets generated automatically for you).

Example 10–1 BenefitLocalHome.java

```
// BenefitLocalHome.java
public interface BenefitLocalHome extends EJBLocalHome
{
   BenefitLocal create() throws CreateException;
   BenefitLocal findByPrimaryKey(Long primaryKey) throws FinderException;
   Collection findAll() throws FinderException;
   BenefitLocal create(Long benefitId, String name) throws CreateException;
   BenefitLocal findByName(String name) throws FinderException;
}
```

The `findAll` method returns a `Collection` of BenefitLocal instances. The `listBenefits` method iterates through the `Collection` and saves the members into an array. It then returns an array of `BenefitModel` instances to the client.

The BenefitModel class is a JavaBean that simply contains the fields for a Benefit object.

```
// BenefitModel.java
package empbft.component.model;
import java.io.Serializable;
public class BenefitModel implements Serializable
{
   private Long _id;
   private String _name;
   private String _description;
   /**
    *  No-Arg constructor required to satisfy contract as a JavaBean. Do
    *  <B>NOT</B> use this constructor.  It will throw a RuntimeException
    *  when used.
    *
    *  @throws RuntimeException
    */
   public BenefitModel()
   {
      throw new RuntimeException(getClass().getName() +
                    ": This is not a valid constructor for this object.");
   }
   /**
    * Constructs a new benefit model containing the details for the benefit.
    */
   public BenefitModel(Long id, String name, String description)
   {
      this._id = id;
      this._name = name;
      this._description = description;
   }

   public String getDescription()   {      return _description;   }
   public void  setDescription(String new_description)
   {
      _description = new_description;
   }

   public Long getId()  {      return _id;  }
   public void setId(Long new_id)   {      _id = new_id;   }
```

```
   public String getName()   {      return _name;  }
   public void setName(String new_name)   {    _name = new_name;  }
}
```

10.4 Details of addNewBenefit Method

To add a new benefit, a client calls the `addNewBenefit` method in the EmployeeBenefitManager session bean. This method calls the `create` method in the BenefitLocalHome interface.

```
// EmployeeBenefitManagerBean.java
public void addNewBenefit(Long benefitId, String benefitName,
                          String benefitDescription)
{
   try
   {
      BenefitLocal newBenefit =
                       getBenefitLocalHome().create(benefitId,benefitName);
      if(benefitDescription!=null)
         newBenefit.setDescription(benefitDescription);
   }
   catch(Exception e)
   {
      e.printStackTrace();
      throw new EJBException("Error adding new benefit item : " + e);
   }
}
```

The `create` method in the BenefitLocalHome interface executes the `ejbCreate` method in the BenefitBean class. In the `ejbCreate` method, you populate the benefit ID and name fields, and return null. The container does the actual work of creating the bean and returning the bean to the caller. The return value type for the `ejbCreate` method is the same as the primary key type for the entity bean.

```
// BenefitBean.java
public Long ejbCreate(Long benefitId, String name)
{
   setBenefitId(benefitId);
   setName(name);
   return null;
}
```

The `addNewBenefit` method then adds a description to the new benefit if the client provided a description.

10.5 Details of listBenefitsOfEmployee Method

To get a list of benefits for a specified employee, a client calls the `listBenefitsOfEmployee` method in the EmployeeBenefitManager session bean. This method calls the `getBenefits` method in the Employee bean, which returns a `Collection` of Benefit local interface objects for the specified employee.

```
// EmployeeBenefitManagerBean.java
public BenefitModel[] listBenefitsOfEmployee(long employeeId)
{
   EmployeeLocal employee = null;
   Collection allBenefits = null;
   BenefitModel benefits[];
   BenefitLocal benefit = null;
   BenefitModel benefitModel = null;

   try
   {
      // Find the employee, then get their benefits
      employee = getEmployeeLocalHome().findByPrimaryKey(new Long(employeeId));
      allBenefits = employee.getBenefits();
      benefits = new BenefitModel[allBenefits.size()];
      Iterator iter = allBenefits.iterator();

      int count = 0;
      while(iter.hasNext())
      {
         benefit = (BenefitLocal)iter.next();
         benefitModel = new BenefitModel(
                        (Long)benefit.getPrimaryKey(),
                        benefit.getName(),
                        benefit.getDescription());
         benefits[count++] = benefitModel;
      }
      return benefits;
   }
   catch(NamingException ne)
   {
      ne.printStackTrace();
      throw new EJBException("Could not find Employee " + employeeId);
   }
```

```
   catch(FinderException fe)
   {
      fe.printStackTrace();
      throw new EJBException("Could not find Employee " + employeeId);
   }
}
```

The `listBenefitsOfEmployee` method takes an employee ID as an input parameter. It calls the `findByPrimaryKey` method in the Employee bean to get the desired Employee instance. It then calls the `getBenefits` method on that instance.

Like other get and set methods, the `getBenefits` method is an abstract method implemented by the EJB container.

The `getBenefits` method returns a `Collection` to the `listBenefitsOfEmployee` method (in EmployeeBenefitManager), which then extracts the contents of the `Collection` into an array of `BenefitModel`'s to return to the client.

Note that the `listBenefitsOfEmployee` and `countEnrollmentsForBenefit` methods (described in the next section) use both the Employee and Benefit entity beans. This requires a relationship field. See Section 10.8, "Relationship Fields in the Entity Beans".

10.6 Details for countEnrollmentsForBenefit Method

To get the number of employees enrolled in a specified benefit, a client calls the `countEnrollmentsForBenefit` method in the EmployeeBenefitManager session bean. This method calls the `getEmployees` method in the Benefit bean.

```
// EmployeeBenefitManagerBean.java
public int countEnrollmentsForBenefit(long benefitId)
{
   try
   {
      BenefitLocal benefit =
                  getBenefitLocalHome().findByPrimaryKey(new Long(benefitId));
      Collection employees = benefit.getEmployees();
      return employees.size();
   }
   catch(Exception e)
   {
      e.printStackTrace();
      throw new EJBException(e);
   }
```

```
}
```

The EmployeeBenefitManager session bean uses the `findByPrimaryKey` method to locate the Benefit bean instance that represents the benefit in question. The `findByPrimaryKey` method is implemented for you by the container. You declare it in the BenefitLocalHome interface (shown in Example 10–1), but do not define it in the BenefitBean class.

The `countEnrollmentsForBenefit` method calls the `getEmployees` method on the instance returned by `findByPrimaryKey` to get a list of employees, and then calls the `size` method to get the number of employees in the list.

Note that the `countEnrollmentsForBenefit` method and `listBenefitsOfEmployee` method (described in the previous section) use both the Employee and Benefit entity beans. This requires a relationship field. See Section 10.8, "Relationship Fields in the Entity Beans".

10.7 Entity Beans and Database Tables

The Employee and Benefit entity beans use container-managed persistence (CMP), which means you have to map the persistent fields in the entity beans to table columns. (You can set the container to do automatic mapping; see the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide* for details.)

Most of the fields in the Employee and the Benefit entity beans map to columns in the EMPLOYEES and BENEFIT tables in the database. Because the beans have different names from the tables, and the names of the fields in the beans do not match the column names, you have to map the names manually in the `ejb-jar.xml` and `orion-ejb-jar.xml` files.

10.7.1 Persistent Fields in the ejb-jar.xml File

In the `ejb-jar.xml` file, the `<cmp-field><field-name>` elements define the persistent fields. For example:

```
// ejb-jar.xml
<entity>
  <description>Entity Bean ( CMP )</description>
  <display-name>Employee</display-name>
  <ejb-name>Employee</ejb-name>

  <local-home>empbft.component.employee.ejb20.EmployeeLocalHome</local-home>
  <local>empbft.component.employee.ejb20.EmployeeLocal</local>
```

```
  <ejb-class>empbft.component.employee.ejb20.EmployeeBean</ejb-class>
  <persistence-type>Container</persistence-type>
  ... lines omitted ...
  <cmp-field><field-name>employeeId</field-name></cmp-field>
  <cmp-field><field-name>firstName</field-name></cmp-field>
  <cmp-field><field-name>lastName</field-name></cmp-field>
  <cmp-field><field-name>emailAddress</field-name></cmp-field>
  <cmp-field><field-name>phoneNumber</field-name></cmp-field>
  <cmp-field><field-name>hireDate</field-name></cmp-field>
  ... lines omitted ...
</entity>

<entity>
   <description>Entity Bean ( CMP )</description>
   <display-name>Benefit</display-name>
   <ejb-name>Benefit</ejb-name>

   <local-home>empbft.component.benefit.ejb20.BenefitLocalHome</local-home>
   <local>empbft.component.benefit.ejb20.BenefitLocal</local>
   <ejb-class>empbft.component.benefit.ejb20.BenefitBean</ejb-class>
   <persistence-type>Container</persistence-type>
   <prim-key-class>java.lang.Long</prim-key-class>
   ... lines omitted ...
   <cmp-field><field-name>benefitId</field-name></cmp-field>
   <cmp-field><field-name>name</field-name></cmp-field>
   <cmp-field><field-name>description</field-name></cmp-field>
   ... lines omitted ...
</entity>
```

10.7.2 Persistent Fields in the Local Interface

In the local interfaces, EmployeeLocal and BenefitLocal, for each persistent field, you declare a pair of accessor methods (get and set methods).

```
// EmployeeLocal.java
package empbft.component.employee.ejb20;
import javax.ejb.EJBLocalObject;
import java.util.Collection;
import java.sql.Timestamp;
public interface EmployeeLocal extends EJBLocalObject
{
   Long getEmployeeId();
   void setEmployeeId(Long newEmployeeId);
   String getFirstName();
```

```
   void setFirstName(String newFirstName);
   String getLastName();
   void setLastName(String newLastName);
   ... lines omitted ...
   Timestamp getHireDate();
   void setHireDate(Timestamp newHireDate);
   ... lines omitted ...
   Collection getBenefits();
   void setBenefits(Collection newBenefits);
}
```

```
// BenefitLocal.java
package empbft.component.benefit.ejb20;
import javax.ejb.EJBLocalObject;
import java.util.Collection;
public interface BenefitLocal extends EJBLocalObject
{
   Long getBenefitId();
   void setBenefitId(Long newBenefitId);
   String getName();
   void setName(String newName);
   String getDescription();
   void setDescription(String newDescription);
   Collection getEmployees();
   void setEmployees(Collection newEmployees);
}
```

10.7.3 Persistent Fields in the orion-ejb-jar.xml File

You map the persistent fields in the entity beans to table columns using the `orion-ejb-jar.xml` file. The following lines from the `orion-ejb-jar.xml` file show the mappings of some fields in the Employee entity bean.

Tip: If you do not want to create the `orion-ejb-jar.xml` file from scratch, let OC4J generate a version of the file. You can then edit the generated file and enter the correct values. To do this:

1. Deploy the application without the `orion-ejb-jar.xml` file. OC4J generates the `orion-ejb-jar.xml` file.
2. Edit the generated `orion-ejb-jar.xml` file, which is located in the `ORACLE_HOME/j2ee/home/application-deployments/<app-name>/<ejb-module-name>` directory.
3. Place the edited file in the same directory as the `ejb-jar.xml` file, and redeploy the application.

```
// orion-ejb-jar.xml
<entity-deployment name="Employee" copy-by-value="false"
                   data-source="jdbc/OracleDS"
                   exclusive-write-access="false" location="Employee"
                   table="EMPLOYEES">
   <primkey-mapping>
      <cmp-field-mapping name="employeeId" persistence-name="EMPLOYEE_ID"
                                           persistence-type="number(6)"/>
   </primkey-mapping>
   <cmp-field-mapping name="firstName" persistence-name="FIRST_NAME"
                                       persistence-type="VARCHAR2(20)"/>
   <cmp-field-mapping name="lastName"  persistence-name="LAST_NAME"
                                       persistence-type="VARCHAR2(25)"/>
   <cmp-field-mapping name="hireDate"  persistence-name="HIRE_DATE"
                                       persistence-type="DATE"/>
   ... lines omitted ...
</entity-deployment>
```

Table 10–2 describes some attributes in the `<entity-deployment>` element.

Table 10–2 Description of Some Attributes in the <entity-deployment> Element

Attribute	Description
`name`	Identifies the entity bean. This value matches the name specified in the `<ejb-name>` element in the `ejb-jar.xml` file.
`table`	Identifies the table in the database.
`location`	Specifies the JNDI name of the entity bean.
`data-source`	Identifies the database. This value refers to the database pointed to in the data-sources.xml file.

The <cmp-field-mapping> element maps fields to columns. The name attribute specifies the field name, and persistence-name specifies the column name.

10.8 Relationship Fields in the Entity Beans

Most of the persistent fields in the Employee and Benefit entity beans map cleanly to corresponding columns in the EMPLOYEES and BENEFITS tables in the database. Also, the countEmployees method accesses the Employee bean only, and the listBenefits and the addNewBenefit methods access the Benefit bean only.

However, the listBenefitsOfEmployee and the countEnrollmentsForBenefit methods use both beans each.

- The listBenefitsOfEmployee method takes an employee ID and returns the benefits selected by the employee. An employee can have zero or more benefits.
- The countEnrollmentsForBenefit method takes a benefit ID and returns the number of employees who are signed up for the benefit. A benefit can have zero or more enrollees.

For these two methods to work, you need to set up a many-to-many relationship between the Employee and Benefit entity beans. The relationship type is many-to-many so that you can look up benefits if you know an employee ID, and you can look up employees if you know a benefit ID.

To determine which employees have which benefits, the many-to-many relationship needs to access the EMPLOYEE_BENEFIT_ITEMS table in the database. This is an association table that enables an employee to have more than one benefit, and a benefit to be associated with more than one employee.

Table 10–3 shows some sample rows in the EMPLOYEE_BENEFIT_ITEMS table. The sample rows show that employee ID 101 has two benefits, and benefit ID 1 has two enrollees.

Table 10–3 Sample Data in EMPLOYEE_BENEFIT_ITEMS Table

EMPLOYEE_ID	BENEFIT_ID	ELECTION_DATE
101	1	1/5/2003
101	2	1/5/2003
102	1	1/6/2003

You set up relationships in the `ejb-jar.xml` file using the `<relationships>` element. Under this parent element, the `<ejb-relation>` element defines each relationship.

Relationships work with fields. The names of the relationship fields are defined in the `<cmr-field-name>` element. In this case, the names of the relationship fields are "benefits" and "employees".

```
// ejb-jar.xml
<relationships>
   <ejb-relation>
      <ejb-relation-name>Employee-Has-Benefits</ejb-relation-name>

      <ejb-relationship-role>
         <ejb-relationship-role-name>Employee-Has-Benefits
                                   </ejb-relationship-role-name>
         <multiplicity>Many</multiplicity>
         <relationship-role-source>
            <ejb-name>Employee</ejb-name>
         </relationship-role-source>
         <cmr-field>
            <cmr-field-name>benefits</cmr-field-name>
            <cmr-field-type>java.util.Collection</cmr-field-type>
         </cmr-field>
      </ejb-relationship-role>

      <ejb-relationship-role>
         <ejb-relationship-role-name>Benefits-Are-For-Employees
                                   </ejb-relationship-role-name>
         <multiplicity>Many</multiplicity>
         <relationship-role-source>
            <ejb-name>Benefit</ejb-name>
         </relationship-role-source>
         <cmr-field>
            <cmr-field-name>employees</cmr-field-name>
            <cmr-field-type>java.util.Collection</cmr-field-type>
         </cmr-field>
      </ejb-relationship-role>
   </ejb-relation>
</relationships>
```

Relationship fields have these features in common with persistent fields:

- You must define get and set methods for the fields in the local interface. The names of the methods are `getBenefits` and `setBenefits` for the Employee

entity bean, and `getEmployees` and `setEmployees` for the Benefit entity bean.

- You map the relationship fields to table columns in the `orion-ejb-jar.xml` file. Example 10–2 shows the mapping for the Employee bean; Example 10–3 shows the mapping for the Benefit bean.

Example 10–2 <entity-deployment> Section for Employee Entity Bean

```
// orion-ejb-jar.xml
<entity-deployment name="Employee" copy-by-value="false"
                       data-source="jdbc/OracleDS"
                       exclusive-write-access="false" location="Employee"
                       table="EMPLOYEES">
   ... other cmp-field-mapping elements omitted ...
   <cmp-field-mapping name="benefits">
      <collection-mapping table="EMPLOYEE_BENEFIT_ITEMS">

         <primkey-mapping>
            <cmp-field-mapping name="employeeId">
               <entity-ref>
                  <cmp-field-mapping name="employeeId"
                                     persistence-name="EMPLOYEE_ID"
                                     persistence-type="NUMBER(6)"/>
               </entity-ref>
            </cmp-field-mapping>
         </primkey-mapping>

         <value-mapping type="empbft.component.benefit.ejb20.BenefitLocal">
            <cmp-field-mapping name="benefitId">
               <entity-ref>
                  <cmp-field-mapping name="benefitId"
                                     persistence-name="BENEFIT_ID"
                                     persistence-type="NUMBER(6)"/>
               </entity-ref>
            </cmp-field-mapping>
         </value-mapping>
      </collection-mapping>
   </cmp-field-mapping>
</entity-deployment>
```

Note the following:

- The `name` attribute for the `<cmp-field-mapping>` element is the same as the relationship name defined in the `ejb-jar.xml` file.

- The `table` attribute for the `<collection-mapping>` element specifies the name of the association table.
- The persistence-name attribute for the `<cmp-field-mapping>` element specifies the name of the primary key column.
- Under the `<primkey-mapping>` element, the `<cmp-field-mapping>` element specifies the name of the foreign key column in the association table that maps to the primary key for the current bean (that is, the Employee bean).
- Under the `<value-mapping>` element, the `<cmp-field-mapping>` element specifies the name of the foreign key column in the association table that maps to the primary key for the target bean (that is, the Benefit bean).

Example 10–3 shows the contents of the `<entity-deployment>` element for the Benefit entity bean. Its contents are similar to that of the Employee bean.

Note that the values in the `<primkey-mapping>` and the `<value-mapping>` elements are reversed from the Employee bean's. The `<primkey-mapping>` element for the Benefit bean specifies the BENEFIT_ID column, and the `<value-mapping>` element specifies the EMPLOYEE_ID column.

Example 10–3 <entity-deployment> Section for the Benefit Entity Bean

```
// orion-ejb-jar.xml
<entity-deployment name="Benefit" copy-by-value="false"
                       data-source="jdbc/OracleDS"
                       exclusive-write-access="false" location="Benefit"
                       table="BENEFITS">
   <primkey-mapping>
      <cmp-field-mapping name="benefitId"
                         persistence-name="BENEFIT_ID"
                         persistence-type="number(6)"/>
   </primkey-mapping>
   <cmp-field-mapping name="name"
                      persistence-name="BENEFIT_NAME"
                      persistence-type="varchar2(50)"/>
   <cmp-field-mapping name="description"
                      persistence-name="BENEFIT_DESCRIPTION"
                      persistence-type="varchar2(255)"/>

   <cmp-field-mapping name="employees">
      <collection-mapping table="EMPLOYEE_BENEFIT_ITEMS">

         <primkey-mapping>
            <cmp-field-mapping name="benefitId">
```

```
                <entity-ref home="Benefit">
                   <cmp-field-mapping name="benefitId"
                                      persistence-name="BENEFIT_ID" />
                </entity-ref>
             </cmp-field-mapping>
          </primkey-mapping>

          <value-mapping type="empbft.component.employee.ejb20.EmployeeLocal">
             <cmp-field-mapping name="employeeId">
                <entity-ref home="Employee">
                   <cmp-field-mapping name="employeeId"
                                      persistence-name="EMPLOYEE_ID" />
                </entity-ref>
             </cmp-field-mapping>
          </value-mapping>
       </collection-mapping>
    </cmp-field-mapping>
 </entity-deployment>
```

11

Enabling Web Services in the Application

Business partners and other clients can access an application's business logic if you expose the business logic as Web Services. These clients include non-Web-based clients, such as standalone Java applications, as well as Web-based clients, such as JSPs.

When you expose the business logic of an application as a Web Service, you specify the methods and their parameters that clients can call. Developers of client applications download the client-side proxy classes ("proxy stubs") for the exposed methods and also the WSDL file, which specifies the names and parameters of the methods. The developers then call the methods provided in the proxy classes to interact seamlessly with the remote Web Service. The proxy classes manage all of the remote interactions with the Web Service, including the marshalling and umarshalling of the Java objects to and from SOAP, and the sending and receiving of the SOAP messages to and from the exposed Web Service location.

This chapter describes how to enable Web Services for the second sample application. It also describes how to write a JSP client application to access the Web Services. The JSP client is part of a different application; the JSP simulates what an external business partner can do to use the exposed Web Services.

For details on Web Services, see the *Oracle Application Server Web Services Developer's Guide*.

Contents of this chapter:

11.1 Enabling Web Services in the Second Sample Application

The second sample application's business logic that we want to expose as Web Services is contained in the `EmployeeBenefitManager` stateless session EJB. Web Services clients can invoke methods listed in its remote interface.

Recall that the `EmployeeBenefitManager` stateless session bean provides the business logic of the sample application. The session bean follows the session facade design pattern, and it uses other EJB components to fulfill its tasks.

The steps to enable Web Services in the sample application are:

1. Check that the data types for the parameters and the return values are valid for Web Services. The *Oracle Application Server Web Services Developer's Guide* lists the valid types.
2. Create a JAR file containing the class files and the deployment descriptors (`ejb-jar.xml` and `orion-ejb-jar.xml`).

   This is a typical JAR file for J2EE applications; it is not different from a typical JAR file because of Web Services.
3. Create a configuration file to provide input for the Web Services Assembly tool, which you will run in the next step. See Section 11.1.1, "Create the Configuration File for the Web Services Assembly Tool" for details.

   The configuration file provides information such as the location of the JAR file, the URL where Web Services clients can access the exposed methods of the sample application, and the name of the EJB that provides the business logic that you want to expose as a Web Service.

   Optionally, you can also provide an additional Java interface object that acts as a marker to identify which methods in the EJB should be exposed in the Web Service. If this is not provided, the Web Services Assembly tool exposes each method in the remote interface of the EJB. The example in this application exposes each method of the session bean, so it does not need to provide the additional Java interface object. See the *Oracle Application Server Web Services Developer's Guide* for more details on the Java interface object.
4. Run the Web Services Assembly tool (`WebServicesAssembler.jar`) to create an EAR file. See Section 11.1.2, "Run the Web Services Assembly Tool".
5. Deploy the EAR file.
6. Test the exposed methods. See Section 11.1.4, "Test the Exposed Methods from the Web Service's Home Page".

11.1.1 Create the Configuration File for the Web Services Assembly Tool

The configuration file `ws-assemble.xml` for the second sample application contains the following lines:

```
<web-service>
  <display-name>Employee Benefit Manager Web Service</display-name>
  <destination-path>build/empbft-ws.ear</destination-path>
  <temporary-directory>.</temporary-directory>
  <context>/employeebenefitmanager</context>

  <stateless-session-ejb-service>
    <path>build/empbft/empbft-ejb.jar</path>
    <uri>/Service</uri>
    <ejb-name>EmployeeBenefitManager</ejb-name>
  </stateless-session-ejb-service>
</web-service>
```

In this case, the file omits the `wsdl-gen` element so that the Web Services Assembly tool does not generate a WSDL file to include in the EAR file. During runtime, when a client requests the WSDL, the OracleAS Web Services runtime generates the WSDL for the client.

Table 11–1 describes the elements in the configuration file. For a complete description of all elements, see the *Oracle Application Server Web Services Developer's Guide*.

Table 11–1 Elements in the Configuration File For the Web Services Assembly Tool

Element	Description
`web-service`	This is the top-level element for the configuration file.
`display-name`	The Web Services Assembly tool uses this value for the `display-name` element in the `application.xml` file.
`destination-path`	This element specifies the name and location of the EAR file generated by the Web Services Assembly tool.
`temporary-directory`	This element specifies the directory where the Web Services Assembly tool can store its temporary files.

Table 11–1 Elements in the Configuration File For the Web Services Assembly Tool

Element	Description
`context`	The Web Services Assembly tool uses this value for the `context-root` element in the `application.xml` file. This element specifies the first part of the URL that clients use to download information about the application's Web Services. The second part of the URL is provided by the `uri` element.
`stateless-session-ejb-service`	This is the parent element for a stateless session bean that is providing Web Services.
`path`	This element specifies the JAR file that contains the stateless session bean.
`uri`	The Web Services Assembly tool copies this value to the `url-pattern` element in the `web.xml` file. This element provides the second part of the URL that clients use to download information about the application's Web Services. This first part of the URL is provided by the `context` element.
`ejb-name`	This element specifies the name of the stateless session bean.

11.1.2 Run the Web Services Assembly Tool

The command for running the Web Services Assembly tool is:

```
prompt> java -jar $ORACLE_HOME/webservices/lib/WebServicesAssembler.jar -config ws-assemble.xml
```

The Web Services Assembly tool does the following:

- Generates Web Service client-side proxy classes for the exposed business logic.
- Sets up an endpoint URL (taken from the `uri` element) for the Web Service.

  You use this URL to access the home page for the Web Service. Developers of client applications also use this URL, with query strings appended, to download the WSDL file and the proxy classes for the Web Service.
- Generates a home page for the Web Services provided by the EJB. The home page contains links where you can test the exposed Web Services.
- Generates the `application.xml` file for the EAR file.

  The Web Services Assembly tool populates this file with values from the `context` and `display-name` elements. It also puts the name of the JAR file

specified in the path element to the `ejb` element in the `application.xml` file. See Figure 11–1.

- Generates the `web.xml` file for the WAR file.

  The Web Services Assembly tool gets the endpoint URL for the Web Service from the `uri` element in the configuration file and puts it in the `web.xml` file. See Figure 11–2.

- Generates an EAR file for the application.

  Table 11–2 describes the contents of the EAR file. The file contains three main files: `empbft-ejb.jar`, `empbft-ws_web.war`, and `application.xml`.

Table 11–2 Files in the Generated EAR File

File	Description
`empbft-ejb.jar`	Contains the class files for the EJB, and two XML files that provide information about the EJB.
`empbft-ws_web.war`	Contains the `index.html` file, which is returned if you access the application using the URL specified in the `application.xml` file. The WAR file also contains the `web.xml` file, which specifies the URL for accessing the home page for the Web Services exposed by the EJB.
`application.xml`	Lists the files that are in the EAR file, and the context root URL for the application.

Figure 11–3 shows the contents of the EAR file.

Figure 11–1 How the Web Services Assembly Tool Generates the application.xml File

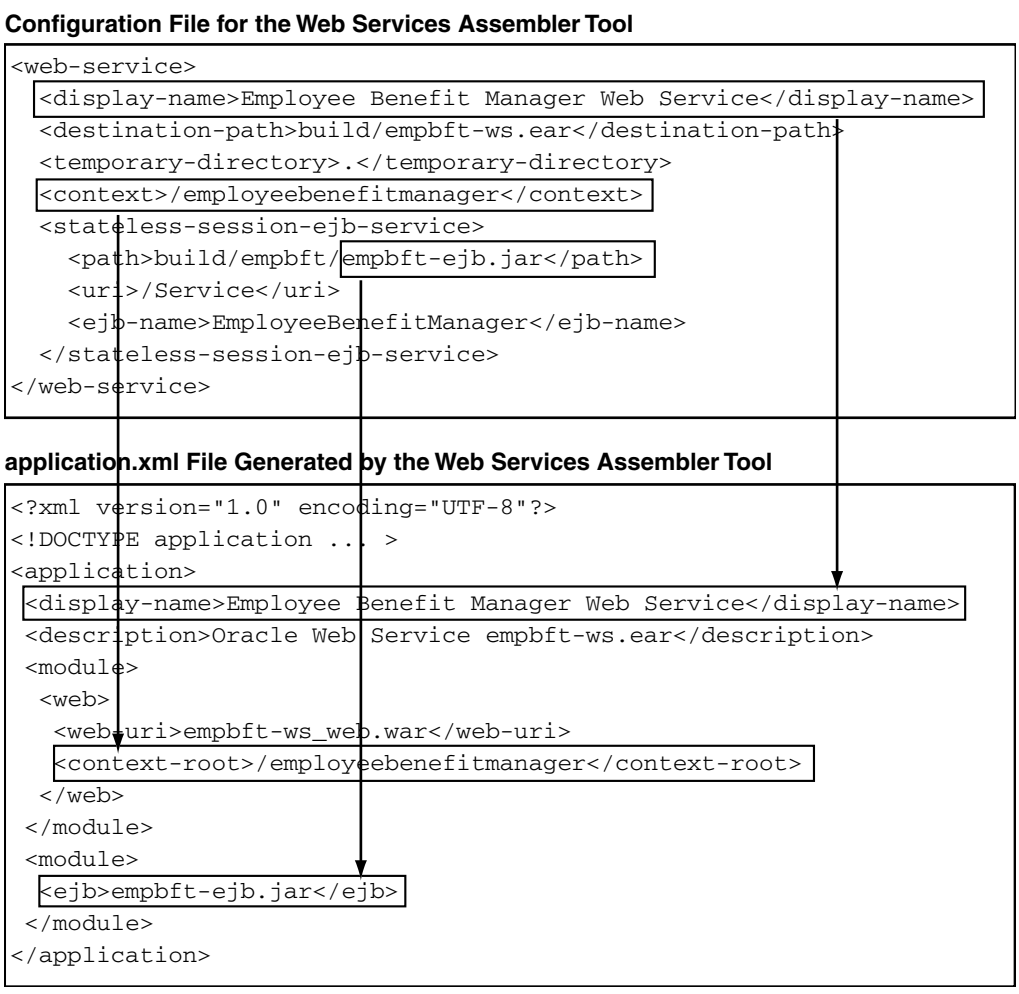
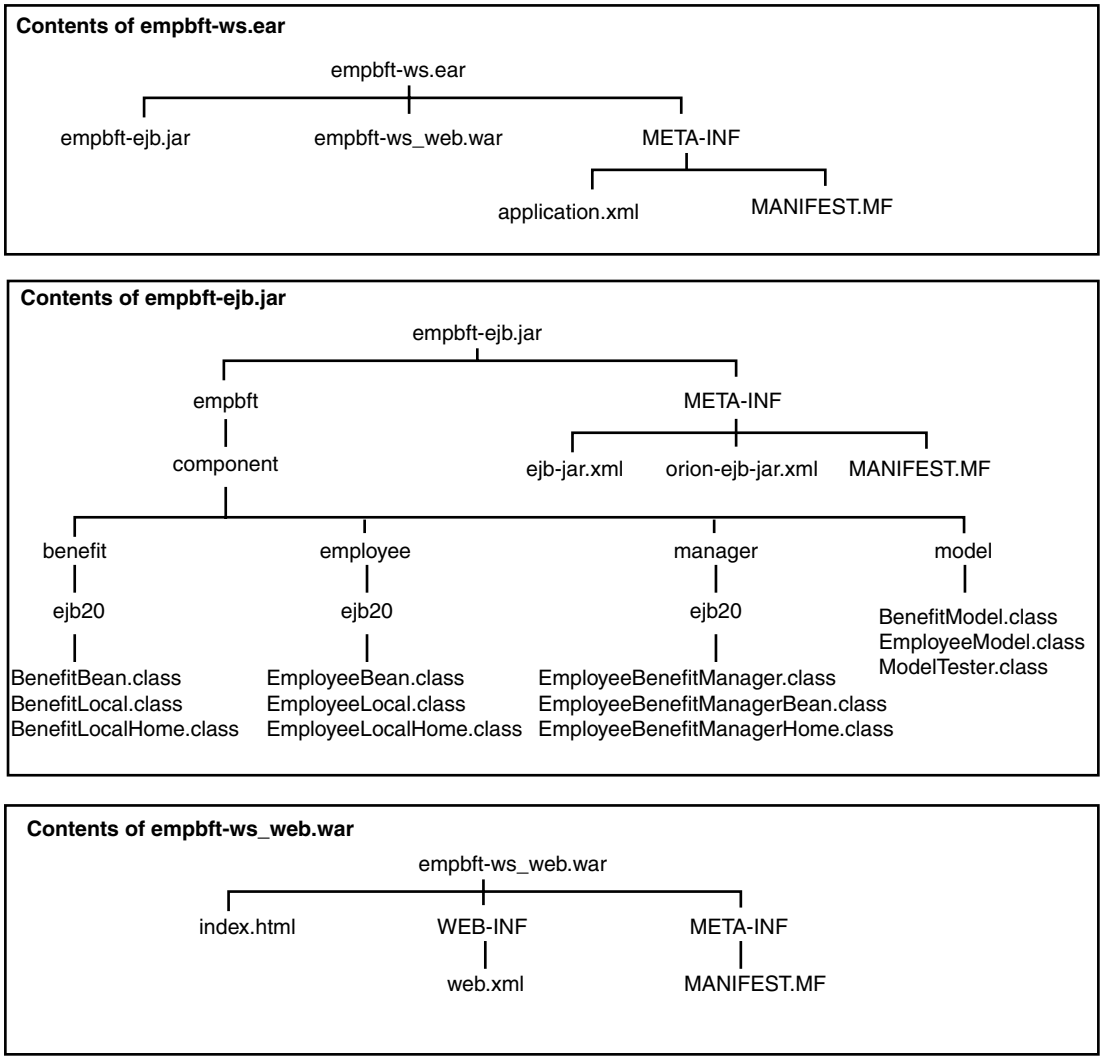


Figure 11–2 How the Web Services Assembly Tool Generates the web.xml File

Figure 11–3 Contents of the Generated EAR File



11.1.3 Deploy the Application

After the Web Services Assembly tool has generated an EAR file, you can deploy the application. The following command deploys the application using the `dcmctl` command.

```
prompt> cd $ORACLE_HOME/dcm/bin
prompt> ./dcmctl deployApplication -file /home/joe/build/empbft-ws.ear -a empbft
```

11.1.4 Test the Exposed Methods from the Web Service's Home Page

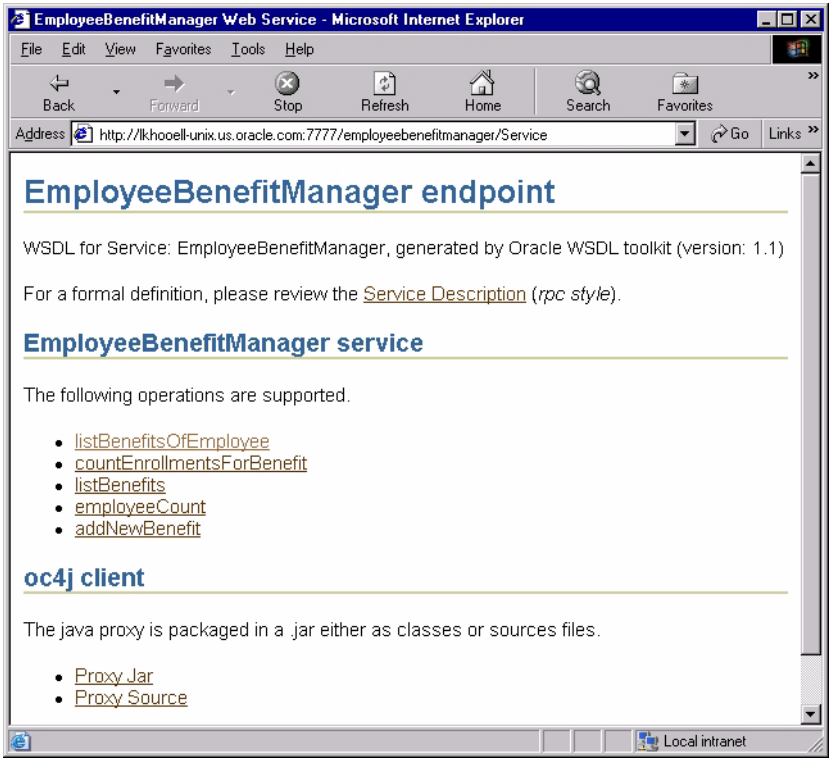
You can test the Web Services from the home page:

1. Invoke the home page for the Web Service.

   The configuration file created in Section 11.1.1, "Create the Configuration File for the Web Services Assembly Tool" specifies the URL for the home page. The URL is the combined values of the `context` and `uri` elements. In this case, the value for `context` is `/employeebenefitmanager`, and the value for `uri` is `/Service`, and thus the URL for the servlet is: `/employeebenefitmanager/Service`.

   Figure 11–4 shows the home page:

Figure 11–4 Home Page for the Web Service

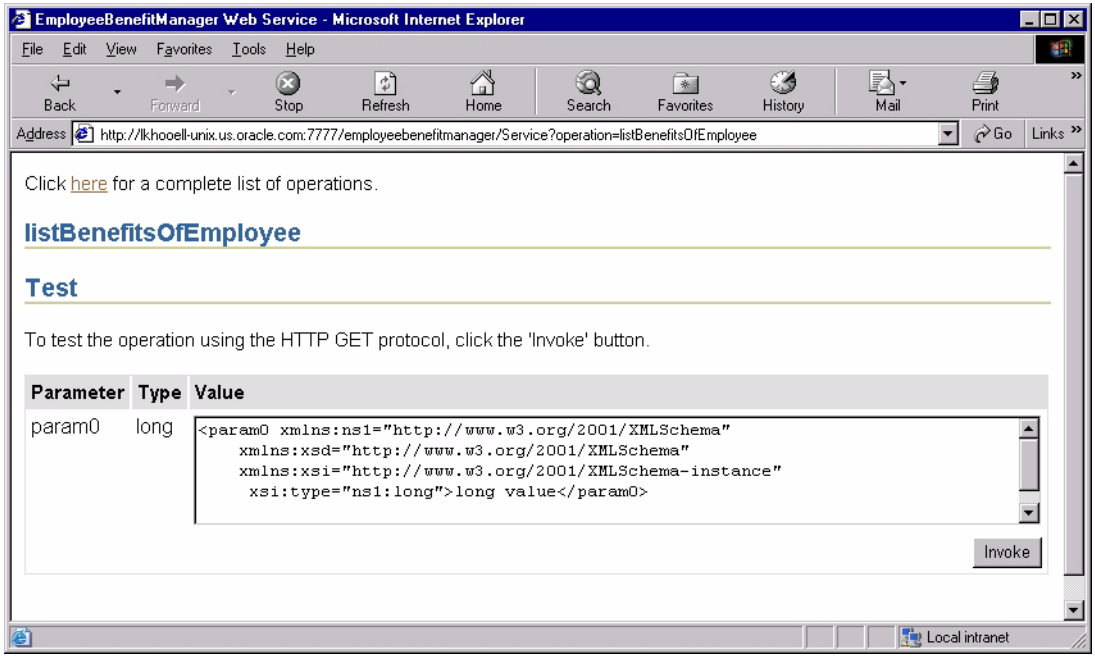


2. Click a method. This displays a page where you can enter parameter values for the method.

3. Enter values for the method's parameters, if any.

   Figure 11–5 shows the parameter page for the `listBenefitsOfEmployee` method. This method requires one parameter: the employee ID. For this method, replace the "long value" text with the actual employee ID value (for example, `188`).

Figure 11–5 Page for Entering Parameter Values

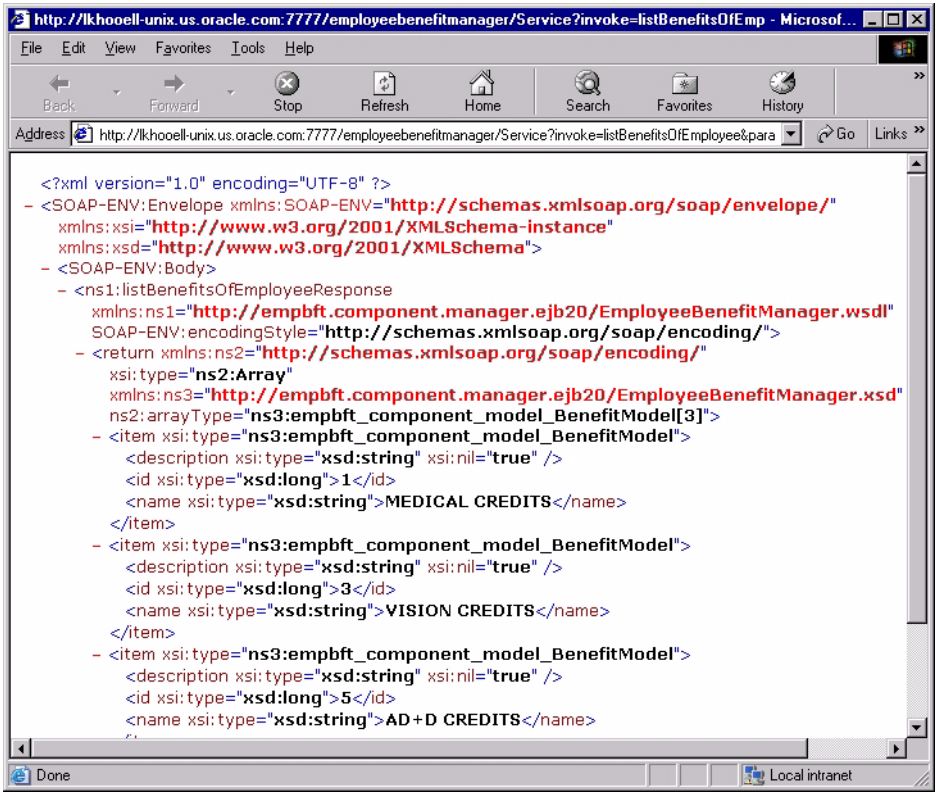


4. Click **Invoke** on the parameter page to invoke the method.

   Figure 11–6 shows the XML returned for the `listBenefitsOfEmployee` method.

   The home page, which is essentially a tool for testing and troubleshooting Web Services, displays the raw XML data. When a client invokes the method, the request goes through proxy classes, which parse the XML data and return only what the method returned. The client does not have to parse the raw XML data.

Figure 11–6 Return Value for the listBenefitsOfEmployee Method



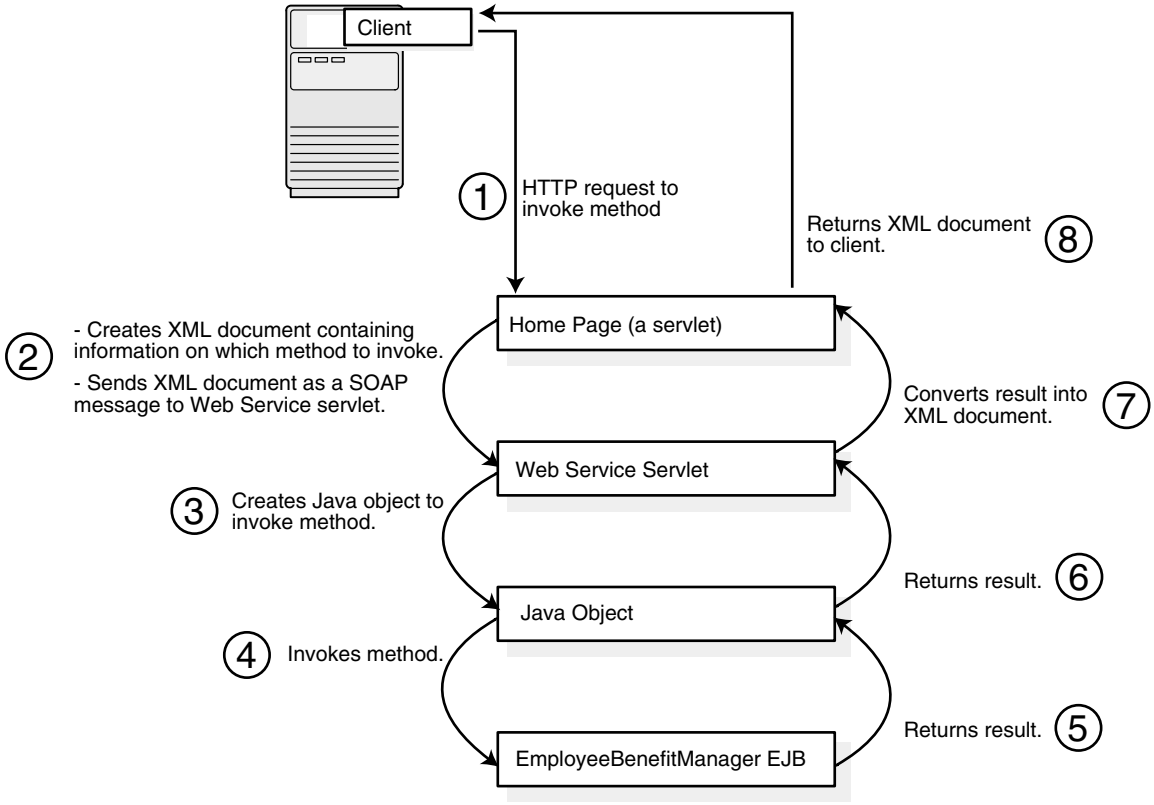
What Happens When You Click Invoke

Figure 11–7 shows what happens when you click **Invoke**.

1. When you click **Invoke**, the browser sends an HTTP request to the home page, which is implemented as a servlet. The request contains information on which method to invoke and parameter values for the method, as necessary.
2. The home page servlet creates an XML document containing the information and sends it as a SOAP message to the OracleAS Web Services servlet.
3. The OracleAS Web Services servlet reads the XML document and creates a Java object to invoke the method.
4. The Java object invokes the method on the specified EJB.
5. The EJB returns the results to the Java object.

6. The Java object returns the results to the OracleAS Web Services servlet.
7. The OracleAS Web Services servlet creates an XML document and inserts the results into the document. It then sends the XML document to the home page servlet.
8. The home page servlet returns the XML document to the client.

Figure 11–7 Request Flow



11.2 Creating a Web Services Client Application

After you have tested the exposed methods in your application, you can allow other developers to access the exposed methods through Web Services. This section describes a client application that invokes the methods provided by the `EmployeeBenefitManager` class in the second sample application.

11.2.1 Design of the Web Services Client

The sample client uses the MVC design pattern, which is described in Chapter 2, "Designing the Application". MVC separates the model (which handles the business logic) from the view (which determines how the data is displayed). The controller directs requests to the proper classes.

The part of the sample client that invokes the Web Services is the model. Classes that make up the model are called handler classes because they handle the business logic processing of requests.

The handler classes invoke the Web Services using generated proxy classes, which are statically bound to the host and application providing the Web Services.

Web Services Clients that Invoke Web Services Dynamically

Instead of creating clients that use the proxy classes, you can create clients that invoke Web Services dynamically. You can even generate your own client from scratch using the WSDL file. For these types of clients, see the *Oracle Application Server Web Services Developer's Guide*.

Other Ways of Designing a Web Services Client

The sample client described in this section is a standard J2EE application. As such, it follows the guidelines for J2EE applications. The only difference is that the client application contains the generated proxy classes in its WAR file.

Web Services also allow for other types of clients, such as JSPs and standalone Java applications. If you have such client applications, then you would follow the guidelines for these types of applications.

Clients can run on the same machine as the application, or on a different machine, or even on a machine that is on a different network.

11.2.2 Steps for Developing a Web Services Client

To develop a client using the generated proxy classes:

1. Download the proxy classes (placed in a zip file) for the `EmployeeBenefitManager` bean from the host where you deployed the sample application created in Section 11.1, "Enabling Web Services in the Second Sample Application".

   The URL to download the proxy classes is:

   ```
   http://<hostname>:<port>/employeebenefitmanager/Service?proxy_jar
   ```

2. Add the ZIP file to your classpath.
3. Download the WSDL file, which describes the exposed methods and the parameters required by the methods.

   The URL to download the WSDL is:

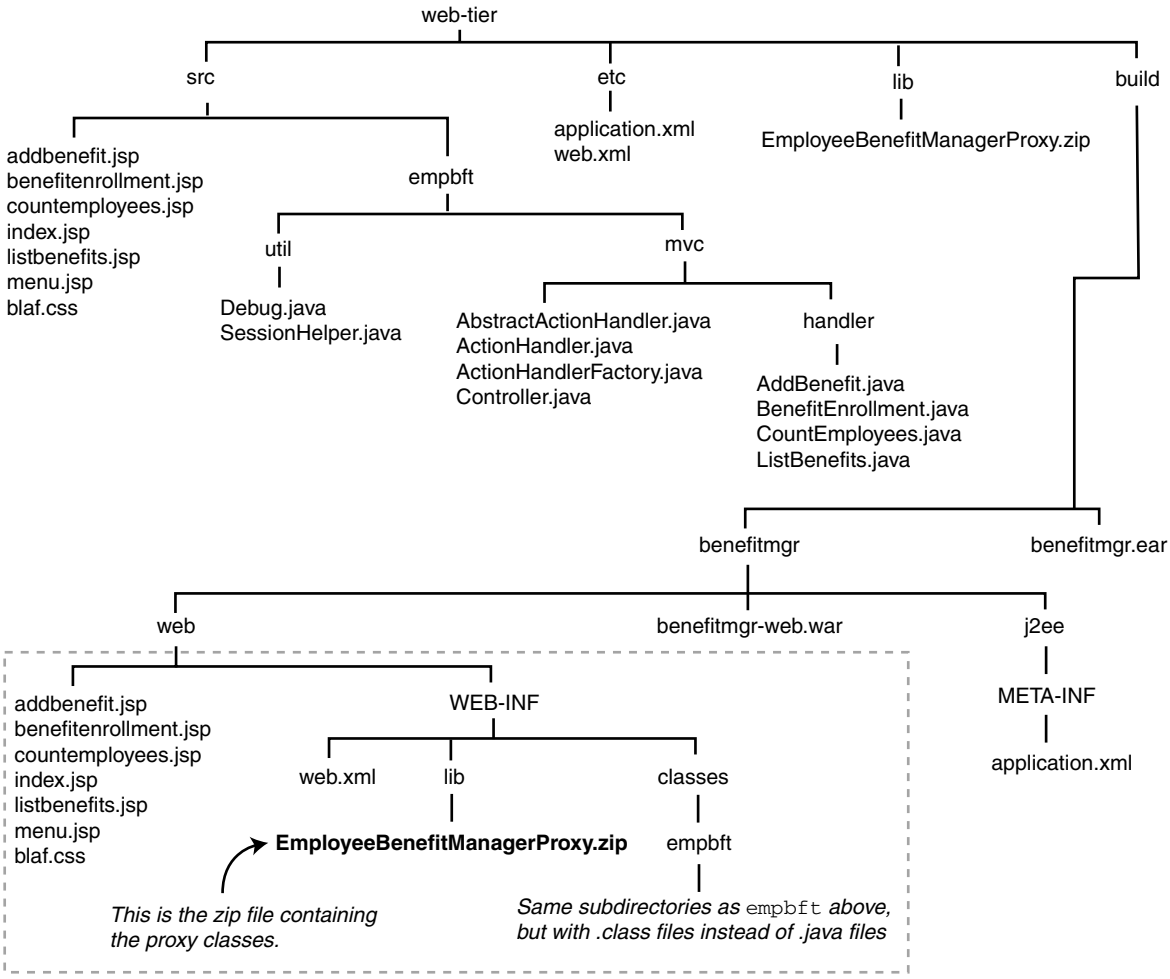
   ```
   http://<hostname>:<port>/employeebenefitmanager/Service?WSDL
   ```

4. Create handler classes to invoke methods in the proxy classes.
5. Create a WAR file that contains the JSP files, the MVC classes, and the downloaded proxy classes. For a Web application, place the proxy classes in the `WEB-INF/lib` directory of the WAR file.
6. Create an EAR file that contains the WAR file and the `application.xml` file.
7. Deploy the EAR file.

11.2.3 Directory Structure for the Web Services Client

Figure 11–8 shows how the files for the Web Services client are organized.

Figure 11–8 Directory Structure for the Web Services Client



The `benefitmgr-web.war` *file contains all the files within the rectangle.*

11.2.4 Request Flow in the Web Services Client

The Web Services client uses the MVC design pattern described in Chapter 2, "Designing the Application". This is how the client responds to a request:

1. A servlet in the Web Services client handles the request initially. It checks the query string of the request to determine which class should handle the request.

2. Handler classes that handle the requests have a method called `performAction`. The servlet invokes this method in the classes.
3. To invoke a method in the proxy stub, a class does the following:
   a. Get an instance of the proxy stub, `EmployeeBenefitManagerProxy` in this case. If an instance does not already exist, create one:

      ```
      EmployeeBenefitManagerProxy proxy = new EmployeeBenefitManagerProxy();
      ```

      In the Web Services client, the `SessionHelper` class instantiates and maintains the proxy class for the duration of the session.

   b. Invoke methods on the proxy class, which will in turn invoke corresponding methods on the remote Web Service. For example:

      ```
      Object[] model = proxy.listBenefits();
      ```

      In the Web Services client, the handler class stores results of the method as an attribute in the request. The handler class then forwards the request to a JSP, which retrieves the JavaBean objects associated with the attribute from the forwarded request and displays the data on a page.

11.2.5 Screens for the Web Services Client

When you invoke the client, you see the following screens:

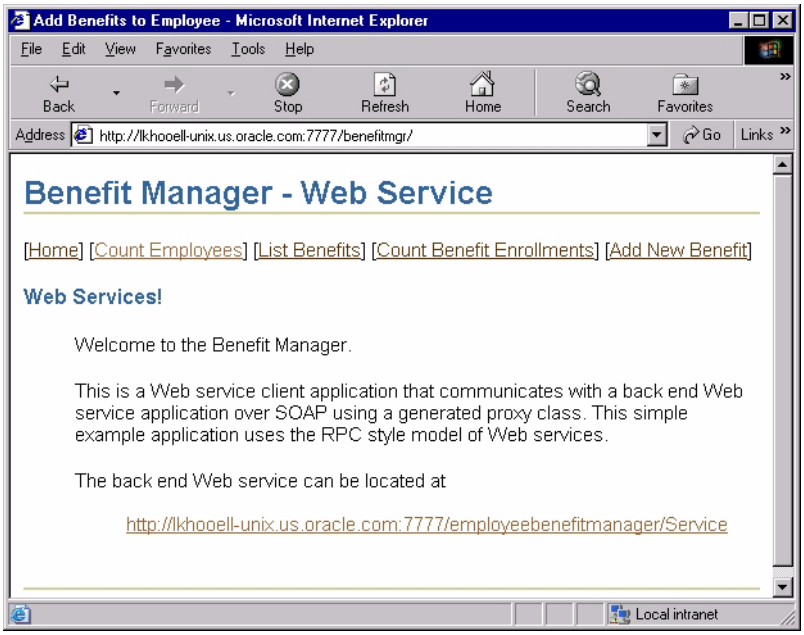
1. Invoke the client. The URL for the client is:

   `http://`*`host`*`:`*`port`*`/benefitmgr`

   Replace *host* with the name of the host where you deployed the client application. Replace *port* with the port number at which the Oracle HTTP Server is listening.

   Figure 11–9 shows the first page of the client.

Figure 11–9 Web Services Client: First Page

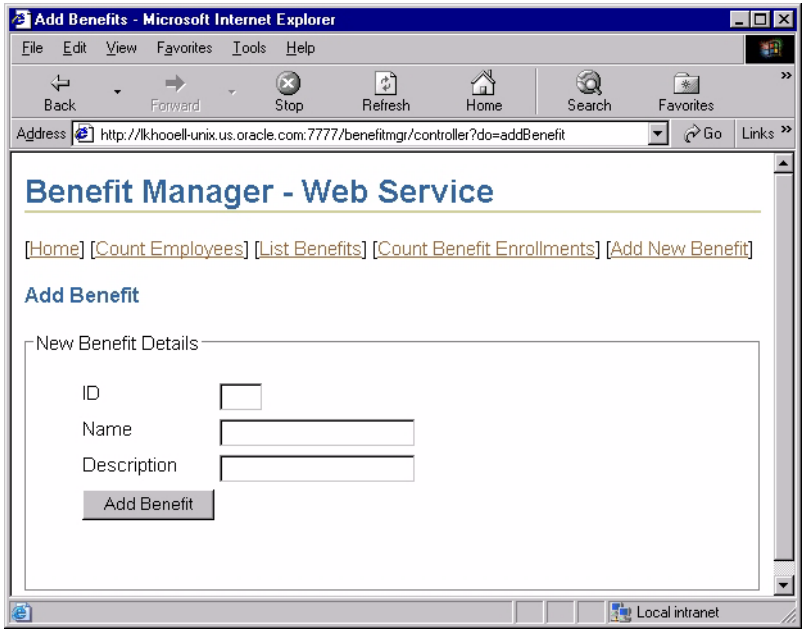


2. Click an item just below the title to invoke a method via Web Services.

   If a method does not require any parameters, the client invokes the method via the proxy class. The proxy class packages the request into proper XML format and sends it to the host. The flow is the same as when you tested the Web Services application. See Figure 11–7 for a diagram of the flow.

   If a method requires parameters, the client displays a page where you can enter values for the parameters. For example, if you click the "Add New Benefit" option, the client displays the following page (Figure 11–10). When you enter the data and click the submit button, the proxy class then dispatches your request (see Figure 11–7 for a diagram of the request flow).

Figure 11–10 Parameter Page for Add New Benefit



A

Configuration Files

This appendix shows the configuration files needed to deploy and run the first sample 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 application. They also define `omsdk`, which is an application that the wireless feature in OracleAS uses.

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

For a detailed description of configuration files, see the *Oracle Application Server Containers for J2EE User's Guide*.

A.1 server.xml

```
<?xml version="1.0"?>
<!DOCTYPE application-server
   PUBLIC "-//Oracle//DTD OC4J Application-server 9.04//EN"
   "http://xmlns.oracle.com/ias/dtds/application-server-09_04.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"?>
<!DOCTYPE web-site PUBLIC "-//Oracle/DTD OC4J Web-site 9.04//EN"
                  "http://xmlns.oracle.com/ias/dtds/web-site-9_04.dtd">
<!-- change the host name below to your own host name. Localhost will -->
<!-- not work with clustering -->
<web-site
       port="9000"
       display-name="Default OracleAS 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-9_04.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

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W